

15A05201 - DATA STRUCTURES

By

**Mr. P. Bhanu Prakash
Associate Professor
DEPT. OF CSE, VEMUIT, CHITTOOR**

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR

II B.Tech II-Sem (E.C.E)

T	Tu	C
3	1	3

(15A05201) DATA STRUCTURES

Objectives:

Understand different Data Structures

Understand Searching and Sorting techniques

Unit-1

Introduction and overview: Asymptotic Notations, One Dimensional array- Multi Dimensional array- pointer arrays.

Linked lists: Definition- Single linked list- Circular linked list- Double linked list- Circular Double linked list- Application of linked lists.

Unit-2

Stacks: Introduction-Definition-Representation of Stack- Operations on Stacks- Applications of Stacks. Queues: Introduction, Definition-Representations of Queues- Various Queue Structures- Applications of Queues. Tables: Hash tables.

Unit-3

Trees: Basic Terminologies- Definition and Concepts- Representations of Binary Tree- Operation on a Binary Tree- Types of Binary Trees-Binary Search Tree, Heap Trees, Height Balanced Trees, B. Trees, Red Black Trees. Graphs: Introduction- Graph terminologies- Representation of graphs- Operations on Graphs- Application of Graph Structures: Shortest path problem- topological sorting.

Unit-4

Sorting : Sorting Techniques- Sorting by Insertion: Straight Insertion sort- List insertion sort- Binary insertion sort- Sorting by selection: Straight selection sort- Heap Sort- Sorting by Exchange- Bubble Sort- Shell Sort-Quick Sort-External Sorts: Merging Order Files-Merging Unorder Files- Sorting Process.

Unit-5

Searching: List Searches- Sequential Search- Variations on Sequential Searches- Binary Search- Analyzing Search Algorithm- Hashed List Searches- Basic Concepts- Hashing Methods- Collision Resolutions- Open Addressing- Linked List Collision Resolution- Bucket Hashing.

Text Books:

1. “Classic Data Structures”, Second Edition by Debasis Samanta, PHI.
2. “Data Structures A Pseudo code Approach with C”, Second Edition by Richard F. Gilberg, Behrouz A. Forouzan, Cengage Learning.

Reference Books:

1. Fundamentals of Data Structures in C – Horowitz, Sahni, Anderson-Freed, Universities Press, Second Edition.
2. Schaum’ Outlines – Data Structures – Seymour Lipschutz – McGrawHill-Revised First Edition.
3. Data structures and Algorithms using C++, Ananda Rao Akepogu and Radhika Raju Palagiri, Pearson Education.

	C225.1 Explain about linked list operations
	C225.2 Interpret the operations of stack and queue
	C225.3 Illustrate concepts of tree and graph
	C225.4 Explain about complexity in sorting technique
	C225.5 Explain time complexities of sorting & searching techniques

WHAT IS DATA STRUCTURE?

A **data structure** is a **data** organization, management and storage format that enable efficient access and modification. A **data structure** is a collection of **data** values, the relationships among them, and the functions or operations that can be applied to the **data**.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned.

DATA: Data can be defined as an elementary value or the collection of value.

For example, student's name and its id are the data about the student.

GROUP ITEMS: Data items which have subordinate data items are called Group item.

For example, name of a student can have first name and the last name.

RECORD: Record can be defined as the collection of various data items.

For example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

FILE: A File is a collection of various records of one type of entity,

For example, if there are 60 students in the class, then there will be 60 records in the related file where each record contains the data about each student.

ATTRIBUTE AND ENTITY: An entity represents the class of certain objects. it contains various attributes.

Each attribute represents the particular property of that entity.

FIELD: Field is a single elementary unit of information representing the attribute of an entity.

NEED OF DATA STRUCTURES

As applications are getting complex and amount of data is increasing day by day, there may arise the following problems:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 10⁶ items in a store, If our application needs to search for a particular item, it needs to traverse 10⁶ items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process. In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

ADVANTAGES OF DATA STRUCTURES

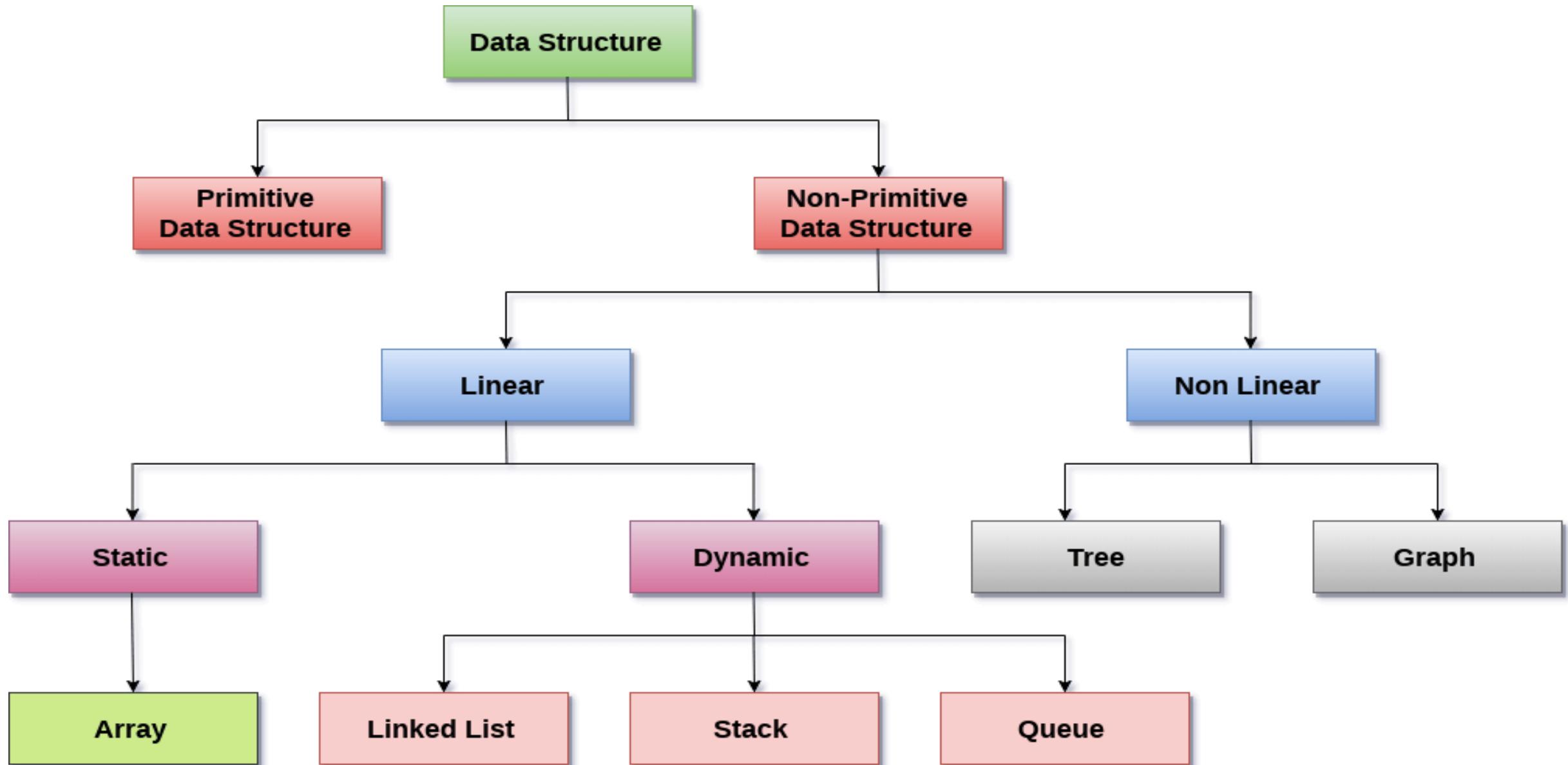
Efficiency: Efficiency of a program depends upon the choice of data structures.

For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. Hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

DATA STRUCTURE CLASSIFICATION



OPERATIONS ON DATA STRUCTURE

1) Traversing: Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subjects, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

2) Insertion: Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is n then we can only insert $n-1$ data elements into it.

3) Deletion: The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then underflow occurs.

4) Searching: The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search.

5) Sorting: The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) Merging: When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size $(M+N)$, then this process is called merging

CHARACTERISTICS OF A DATA STRUCTURE

- **Correctness** – Data structure implementation should implement its interface correctly.
- **Time Complexity** – Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** – Memory usage of a data structure operation should be as little as possible.

ALGORITHM

An algorithm is a procedure having well defined steps for solving a particular problem. Algorithm is finite set of logic or instructions, written in order for accomplish the certain predefined task. It is not the complete program or code, it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudo code.

The major categories of algorithm are :

- **Sort:** Algorithm developed for sorting the items in certain order.
- **Search:** Algorithm developed for searching the items inside a data structure.
- **Delete:** Algorithm developed for deleting the existing element from the data structure.
- **Insert:** Algorithm developed for inserting an item inside a data structure.
- **Update:** Algorithm developed for updating the existing element inside a data structure.t

The performance of algorithm is measured on the basis of following properties:

Time complexity: It is a way of representing the amount of time needed by a program to run to the completion.

Space complexity: It is the amount of memory space required by an algorithm, during a course of its execution.

Space complexity is required in situations when limited memory is available and for the multi user system.

Each algorithm must have:

Specification: Description of the computational procedure.

Pre-conditions: The condition(s) on input.

Body of the Algorithm: A sequence of clear and unambiguous instructions.

Post-conditions: The condition(s) on output.

CHARACTERISTICS OF AN ALGORITHM

An algorithm must follow the mentioned below characteristics:

- **Input:** An algorithm must have 0 or well defined inputs.
- **Output:** An algorithm must have 1 or well defined outputs, and should match with the desired output.
- **Feasibility:** An algorithm must be terminated after the finite number of steps.
- **Independent:** An algorithm must have step-by-step directions which is independent of any programming code.
- **Unambiguous:** An algorithm must be unambiguous and clear. Each of their steps and input/outputs must be clear and lead to only one meaning.

ASYMPTOTIC NOTATION

Asymptotic Notation:

A problem may have numerous (many) algorithmic solutions. In order to choose the best algorithm for a particular task, you need to be able to judge how long a particular solution will take to run.

Asymptotic notation of an algorithm is a mathematical representation of its complexity

Asymptotic notation is used to judge the best algorithm among numerous algorithms for a particular problem.

Asymptotic complexity is a way of expressing the main component of algorithms like

- Cost
- Time complexity
- Space complexity

Some Asymptotic notations are

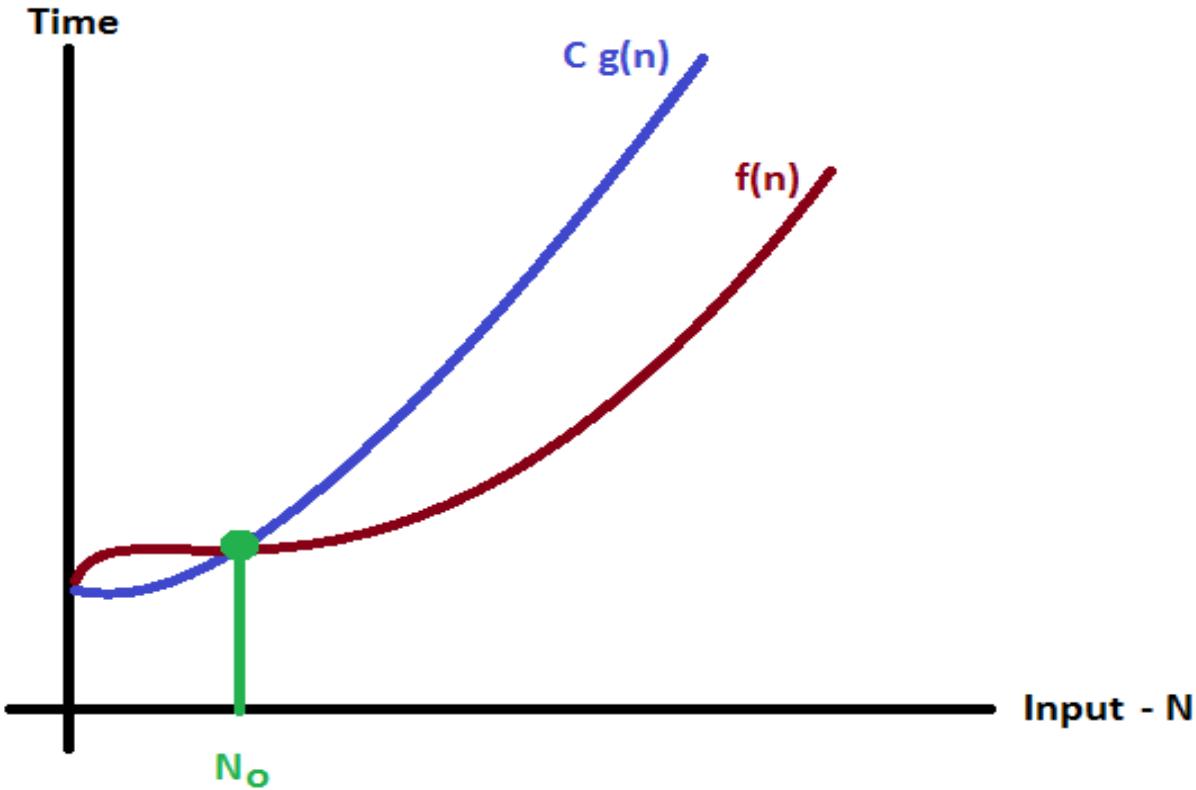
1. Big oh $\rightarrow O$
2. Omega $\rightarrow \Omega$
3. Theta $\rightarrow \theta$
4. Little oh $\rightarrow o$
5. Little Omega $\rightarrow \omega$

1. Big - Oh Notation (O)

- Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
- That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

The function $f(n) = O(g(n))$ (read as “f of n is big oh of g of n) iff (if and only if) there exit positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all $n, n \geq n_0$ $f(n) = O(g(n))$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C g(n)$ is greater than $f(n)$ which indicates the algorithm's upper bound.

Example

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\mathbf{O}(g(n))$ then it must satisfy $f(n) \leq C \times g(n)$ for all values of $C > 0$ and $n_0 \geq 1$

$$f(n) \leq C g(n)$$

$$\Rightarrow 3n + 2 \leq C n$$

Above condition is always TRUE for all values of $C = 4$ and $n \geq 2$.

By using Big - Oh notation we can represent the time complexity as follows...

$$3n + 2 = \mathbf{O}(n)$$

Big - Omega Notation (Ω)

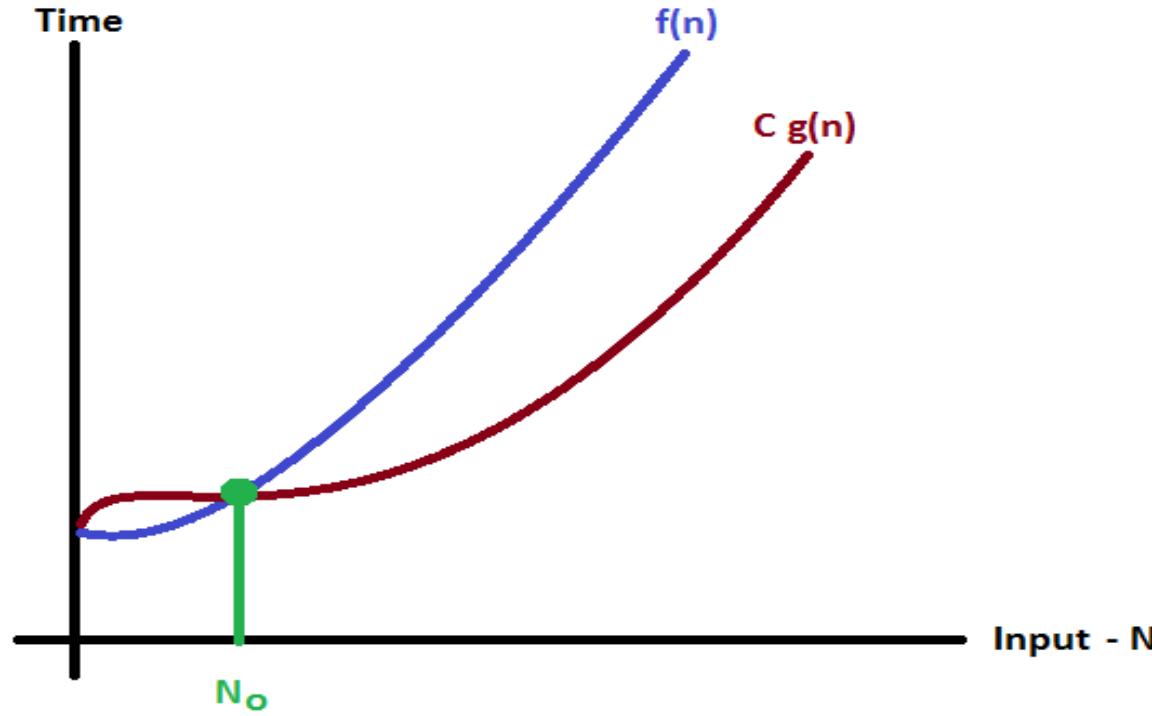
- Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
- That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

The function $f(n) = \Omega(g(n))$ (read as “f of n is omega of g of n) iff (if and only if) there exit positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all $n, n \geq n_0$

$$f(n) = \Omega(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C \times g(n)$ is less than $f(n)$ which indicates the algorithm's lower bound.

EXAMPLE

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\Omega(g(n))$ then it must satisfy $f(n) \geq C g(n)$ for all values of $C > 0$ and $n_0 \geq 1$

$$f(n) \geq C g(n)$$

$$\Rightarrow 3n + 2 \leq C n$$

Above condition is always TRUE for all values of $C = 1$ and $n \geq 1$.

By using Big - Omega notation we can represent the time complexity as follows...

$$3n + 2 = \Omega(n)$$

Big - Theta Notation (Θ)

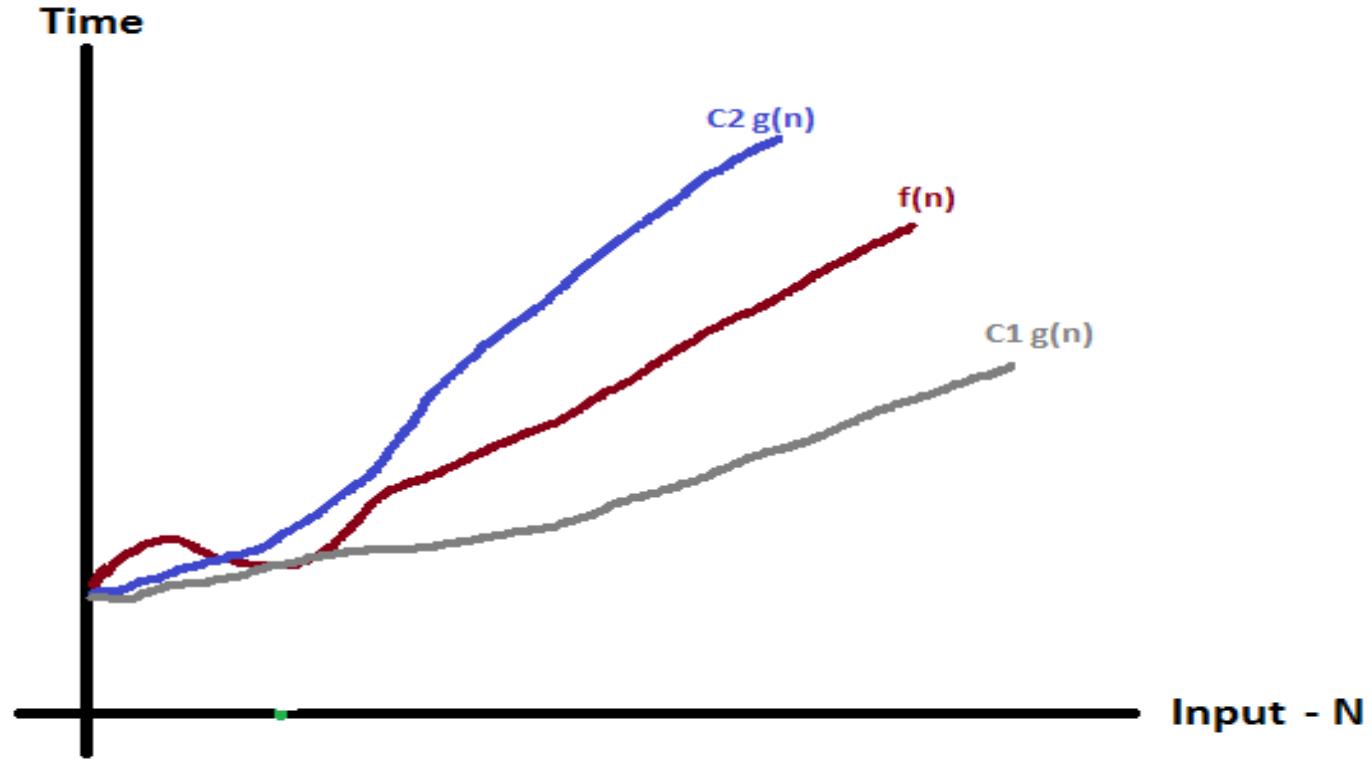
- Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.
- That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

The function $f(n) = \Theta(g(n))$ (read as “f of n is theta of g of n) iff (if and only if) there exist positive constants c_1, c_2 and n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n, n \geq n_0$

$$f(n) = \Theta(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C_1 g(n)$ is less than $f(n)$ and $C_2 g(n)$ is greater than $f(n)$ which indicates the algorithm's average bound.

EXAMPLE

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\Theta(g(n))$ then it must satisfy $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all values of C_1 , $C_2 > 0$ and $n_0 \geq 1$

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

$$C_1 n \leq 3n + 2 \leq C_2 n$$

Above condition is always TRUE for all values of $C_1 = 1$, $C_2 = 4$ and $n \geq 1$.

By using Big - Theta notation we can represent the time complexity as follows...

$$3n + 2 = \Theta(n)$$

Little oh: o

The function $f(n)=o(g(n))$ (read as “f of n is little oh of g of n”) iff

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \quad \text{for all } n, n \geq 0$$

Example:

$3n+2= o(n^2)$ as

$$\lim_{n \rightarrow \infty} \frac{(3n+2)}{n^2} = 0$$

Little Omega: ω

The function $f(n)=\omega(g(n))$ (read as “f of n is little omega of g of n”) iff

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0 \quad \text{for all } n, n \geq 0$$

Example:

$3n+2= o(n^2)$ as

$$\lim_{n \rightarrow \infty} \frac{n^2}{(3n+2)} = \infty$$

Arrays

OPERATIONS ON ARRAYS

- ✓ Traversing an array
- ✓ Inserting an element in an array
- ✓ Searching an element in an array
- ✓ Deleting an element from an array
- ✓ Merging two arrays
- ✓ Sorting an array in ascending or descending order

Algorithm for array traversal

step 1: [initialization] set $i = \text{lower_bound}$

step 2: repeat steps 3 to 4 while $i \leq \text{upper_bound}$

step 3: apply process to $a[i]$

step 4: set $i = i + 1$

[end of loop]

step 5: exit

Inserting an Element in an Array

Algorithm to insert a new element to the end of an array

Step 1: Set upper_bound = upper_bound + 1

Step 2: Set A[upper_bound] = VAL

Step 3; EXIT

EXAMPLE

Data []={12, 23, 34, 45, 56, 67, 78, 89, 90,100};

- a) Calculate the length of the array
- b) Find the lower and upper-bound
- c) Show the memory representation of the array
- d) If the new data element with the value 75 has to be inserted find its position
- e) Insert the 75 and show the memory representation

SOLUTION:

Length of the array =number of the elements

Lower bound= 0 upper bound = 9

Index	Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]
Data value	12	23	34	45	56	67	78	89	90	100

Blue

Yellow

Red

“ Arrays” are one type of data structure, and can store multiple values

a[0]

a[1]

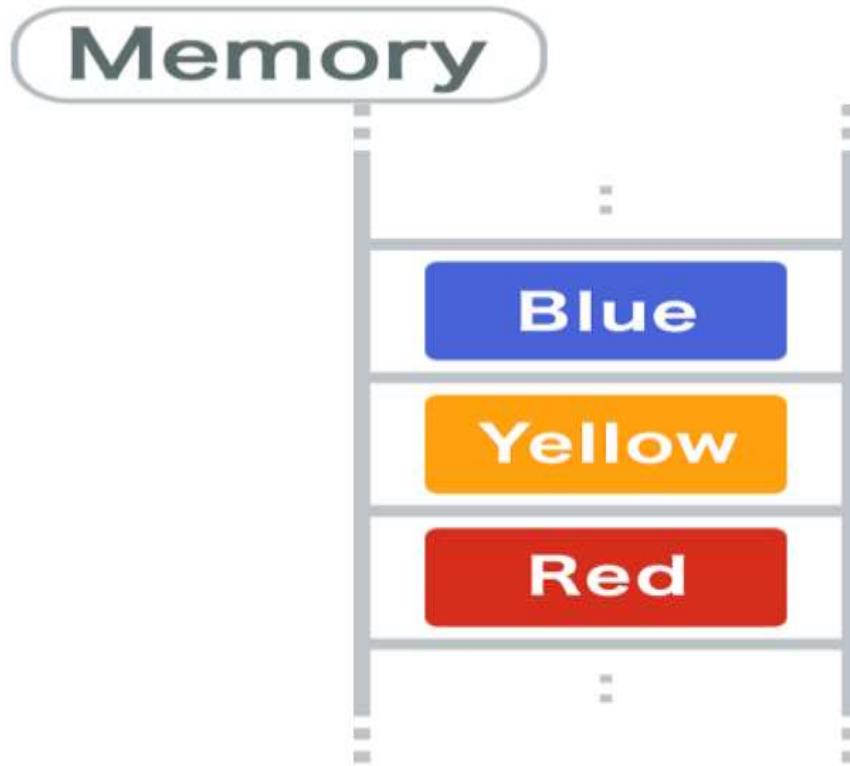
a[2]

Blue

Yellow

Red

Each element can be accessed through its index (a number that denotes its order with in the data).



Data is stored sequentially in memory in consecutive locations

Random Access

a[0]

Blue

a[1]

Yellow

a[2]

Red

Because they're stored in consecutive locations, memory address can be calculated using their indices, allowing for random access of data.

a[0]

Blue

a[1]

Yellow

a[2]

Red

Another feature of arrays is that adding or deleting data in a specific location carries a high cost compared to lists.

a[0]

Blue

a[1]

Yellow

a[2]

Red

Green

Let's imagine adding "Green" to the 2nd location

a[0]

Blue

a[1]

Yellow

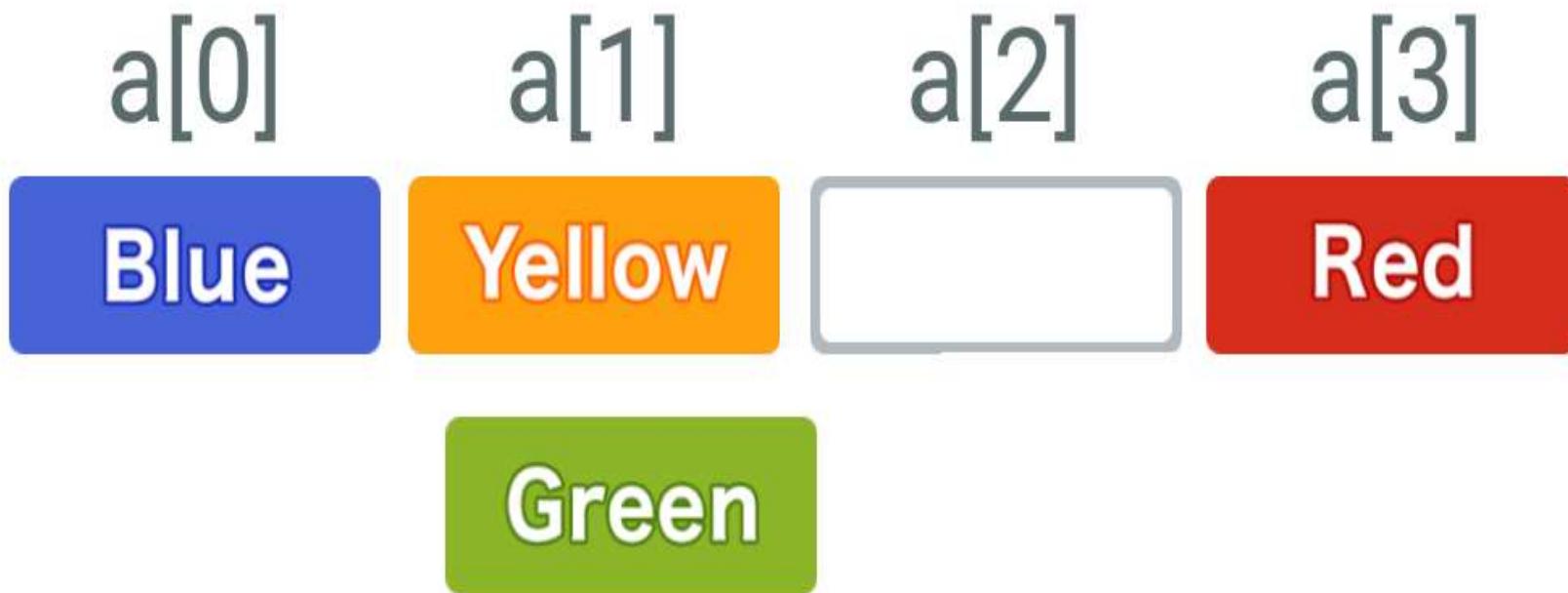
a[2]

Red

a[3]

Green

First we secure an additional space at the end of the array.



In order to free up the space needed for the addition, data is shifted one element at a time.



In order to free up the space needed for the addition, Data is shifted one element at a time.



“Green” is added to the empty space, completing the addition.

Conversely, when deleting the second element,

INSERTION

Algorithm INSERT(A,POS, VAL) to insert an element VAL at position POS

Step 1: [INITIALIZATION] SET I = N

Step 2: Repeat Steps 3 and 4 while I >= POS

Step 3: SET A[I + 1] = A[I]

Step 4: SET I = I – 1

[End of Loop]

Step 5: SET N = N + 1

Step 6: SET A[POS] = VAL

Step 7: EXIT

Initial data array

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
45	23	34	12	56	20

Calling Insert (Data, 3, 100) will lead to the following in the array

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]
45	23	34	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]
45	23	34	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]
45	23	34	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]
45	23	34	100	12	56	20

Deleting an Element from an Array

Algorithm to delete an element from the end of the array

```
Step 1: Set upper_bound = upper_bound - 1  
Step 2: EXIT
```

Algorithm DELETE(A, POS) to delete an element at POS

```
Step 1: [INITIALIZATION] SET I = POS  
Step 2: Repeat Steps 3 and 4 while I <= N-1  
Step 3: SET A[I] = A[I + 1]  
Step 4: SET I = I + 1  
        [End of Loop]  
Step 5: SET N = N - 1  
Step 6: EXIT
```



We first delete the element.

a[0]

a[1]

a[2]

a[3]

Blue

Yellow



Red

Green

Then fill in the empty space by shifting
the data one element at a time.



Then fill in the empty space by shifting
the data one element at a time.



Finally, the extra space is deleted completing the deletion.

INITIAL DATA ARRAY

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
45	23	34	12	56	20

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
45	23	12	12	56	20

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
45	23	12	56	56	20

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]
45	23	12	56	20	20

Data[0]	Data[1]	Data[2]	Data[3]	Data[4]
45	23	12	56	20

MERGING OF TWO ARRAYS

Array 1	90	56	89	77	69
---------	----	----	----	----	----

Array 2	45	88	76	99	12	58	81
---------	----	----	----	----	----	----	----

Array 3	90	56	89	77	69	45	88	76	99	12	58	81
---------	----	----	----	----	----	----	----	----	----	----	----	----

Merging of two unsorted arrays

If we have two sorted arrays and the resultant merged array needs to be a sorted one, then the task of merging the arrays become a little difficult.

Array 1	20	30	40	50	60
---------	----	----	----	----	----

Array 2	15	22	31	45	56	62	78
---------	----	----	----	----	----	----	----

Array 3	15	20	22	30	31	40	45	50	56	60	62	78
---------	----	----	----	----	----	----	----	----	----	----	----	----

Merging of two sorted arrays

Applications of Arrays

- Arrays are widely used to implement mathematical vectors, matrices and other kinds of rectangular tables.
- Many databases include one-dimensional arrays whose elements are records.
- Arrays are also used to implement other data structures like heaps, hash tables, deques, queues, stacks and string. We will read about these data structures in the subsequent chapters.
- Arrays can be used for dynamic memory allocation.

Linked list

Basic Operations





"Lists" are one type of data structure, and can store multiple values.

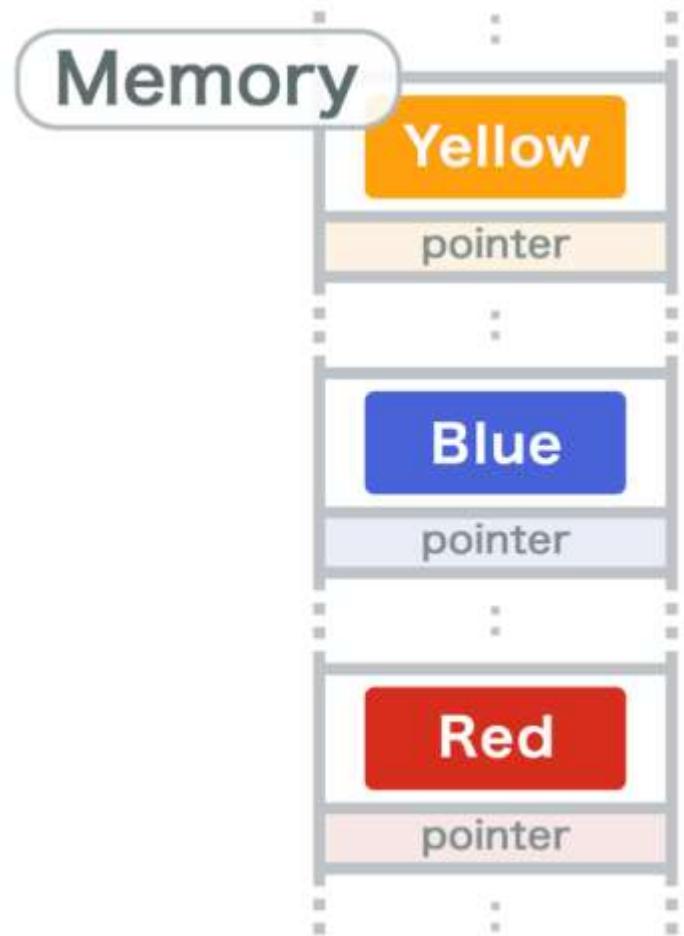


Pointer



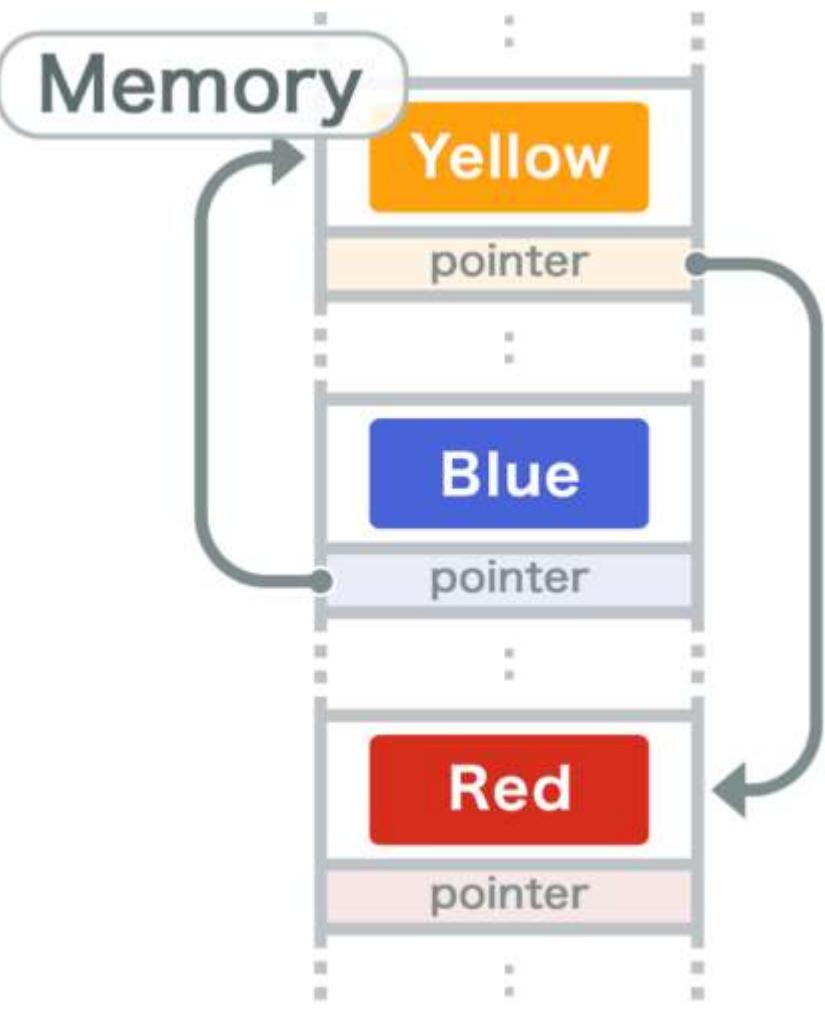
They are unique in how they pair data with "pointers", the pointers indicating the next piece of data's memory location.





In lists, data is stored in various disjointed locations in memory.





In lists, data is stored in various disjointed locations in memory.



Sequential Access



Because data is stored in different locations, each piece of data can only be accessed through the pointer that precedes it.



Sequential Access



Because data is stored in different locations, each piece of data can only be accessed through the pointer that precedes it.



Sequential Access



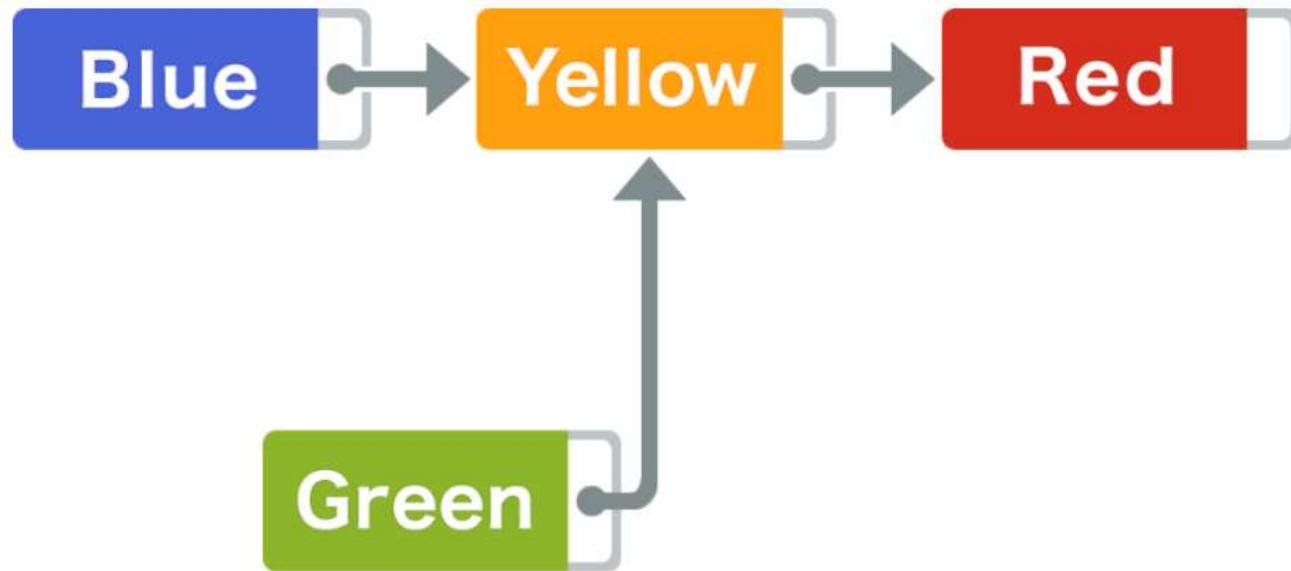
Because data is stored in different locations, each piece of data can only be accessed through the pointer that precedes it.





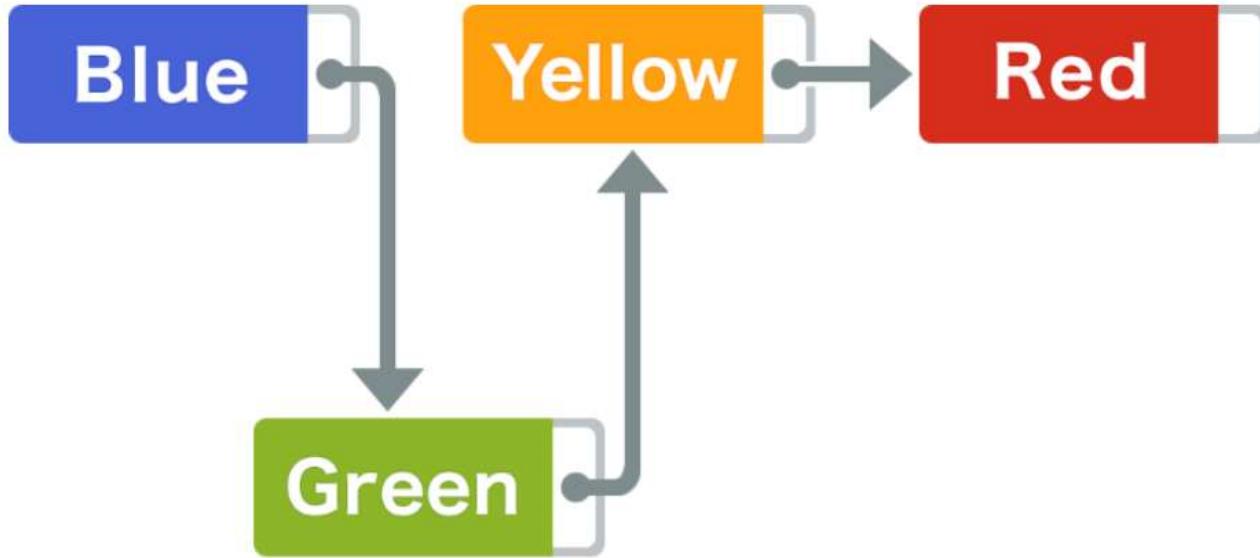
Addition of data is performed simply by replacing the pointers on either side of the addition.





Addition of data is performed simply by replacing the pointers on either side of the addition.

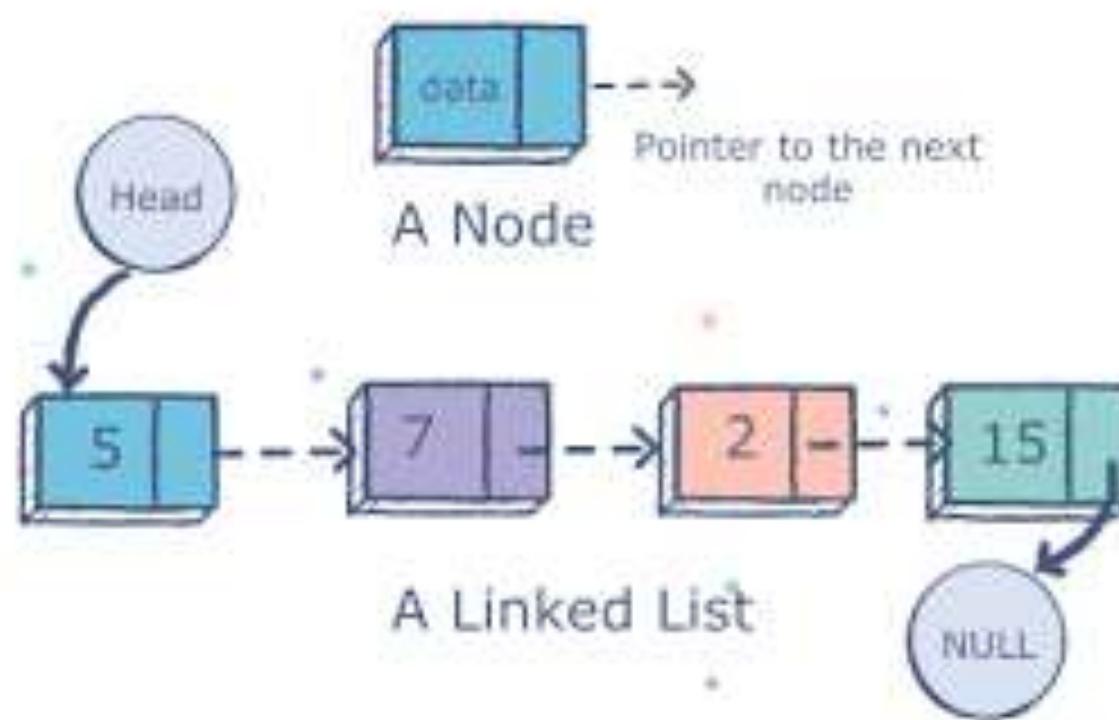




Addition of data is performed simply by replacing the pointers on either side of the addition.



Linked Lists



Algorithm for traversing a linked list

Step 1: [Initialize] Set Ptr = Start

Step 2: Repeat Steps 3 And 4 While Ptr != Null

Step 3: Apply Process To Ptr -> Data

Step 4: Set Ptr = Ptr -> Next

[End Of Loop]

Step 5: Exit

Algorithm to print the number of nodes in a linked list

Step 1: [Initialize] Set Count =0

Step 2: [Initialize] Set Ptr = Start

Step 3: Repeat Teps 4 And 5 While Ptr != Null

Step 4: Set Count = +1

Step 5: Set Ptr = Ptr -> Next

[End Of Loop]

Step 6: Write Count

Step 7: Exit

Algorithm to search a linked list

Step 1: [Initialize] Set Ptr = Start

Step 2: Repeat Step 3 While Ptr != Null

Step 3: If Val = Ptr ->Data

Set Pos = Ptr

Go To Step 5

Else

Set Ptr = Ptr -> Next

[End Of If]

[End Of Loop]

Step 4: Set Pos = Null

Step 5: Exit

Inserting a New Node in a Linked List

Case 1: **The new node is inserted at the beginning.**

Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

Case 4: The new node is inserted before a given node.

Step 1: If Avail = Null

 Write Overflow

 Go To Step 7

 [End Of If]

Step 2: Set New_node = Avail

Step 3: Set Avail = Avail -> Next

Step 4: Set New_node ->Data = Val

Step 5: Set New_node -> Next = Start

Step 6: Set Start = New_node

Step 7: Exit

Algorithm new node is inserted at the end

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 10
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Algorithm new node is inserted after a given node.

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 12
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

Algorithm new node is inserted before a given node.

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 12
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR -> DATA != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

Algorithm to delete the first node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Algorithm to delete the last node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 8
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR->NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR->NEXT
    [END OF LOOP]
Step 6: SET PREPTR->NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Algorithm to delete the node after a given node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 10
    [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR->DATA != NUM
Step 5:     SET PREPTR = PTR
Step 6:     SET PTR = PTR->NEXT
    [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR->NEXT = PTR->NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

CIRCULAR LINKED LISTS

Inserting a New Node in a Circular Linked List

Case 1: The new node is inserted at the beginning of the circular linked list.

Case 2: The new node is inserted at the end of the circular linked list.

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 11
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     PTR = PTR -> NEXT
    [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

Algorithm new node is inserted at the end of the circular linked list

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 10
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
| Step 8:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Algorithm to delete the first node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR → NEXT != START
Step 4:         SET PTR = PTR → NEXT
        [END OF LOOP]
Step 5: SET PTR → NEXT = START → NEXT
Step 6: FREE START
Step 7: SET START = PTR → NEXT
Step 8: EXIT
```

Algorithm to delete the last node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 8
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR->NEXT != START
Step 4:         SET PREPTR = PTR
Step 5:         SET PTR = PTR->NEXT
    [END OF LOOP]
Step 6: SET PREPTR->NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```

DOUBLY LINKED LISTS

Algorithm to insert a new node at the beginning

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 9
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

Algorithm to insert a new node at the end

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 11
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

Algorithm to insert a new node after the given node

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 12
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

Algorithm to insert a new node before the given node

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 12
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR
Step 9: SET NEW_NODE -> PREV = PTR -> PREV
Step 10: SET PTR -> PREV = NEW_NODE
Step 11: SET PTR -> PREV -> NEXT = NEW_NODE
Step 12: EXIT
```

Deleting a Node from a Doubly Linked List

- Case 1: The first node is deleted.
- Case 2: The last node is deleted.
- Case 3: The node after a given node is deleted.
- Case 4: The node before a given node is deleted.

Algorithm to delete the first node

Step 1: IF START = NULL
 Write UNDERFLOW
 Go to Step 6
 [END OF IF]

Step 2: SET PTR = START
Step 3: SET START = START → NEXT
Step 4: SET START → PREV = NULL
Step 5: FREE PTR
Step 6: EXIT

Algorithm to delete the last node

```
Step 1: IF START = NULL  
        Write UNDERFLOW  
        Go to Step 7  
    [END OF IF]  
Step 2: SET PTR = START  
Step 3: Repeat Step 4 while PTR->NEXT != NULL  
Step 4:     SET PTR = PTR->NEXT  
    [END OF LOOP]  
Step 5: SET PTR->PREV->NEXT = NULL  
Step 6: FREE PTR  
Step 7: EXIT
```

Algorithm to delete a node after a given node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 9
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR → DATA != NUM
Step 4:     SET PTR = PTR → NEXT
    [END OF LOOP]
Step 5: SET TEMP = PTR → NEXT
Step 6: SET PTR → NEXT = TEMP → NEXT
Step 7: SET TEMP → NEXT → PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

Algorithm to delete a node before a given node

```
Step 1: IF START = NULL
        Write UNDERFLOW
        Go to Step 9
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->DATA != NUM
Step 4:     SET PTR = PTR->NEXT
    [END OF LOOP]
Step 5: SET TEMP = PTR->PREV
Step 6: SET TEMP->PREV->NEXT = PTR
Step 7: SET PTR->PREV = TEMP->PREV
Step 8: FREE TEMP
Step 9: EXIT
```

CIRCULAR DOUBLY LINKED LISTS

Algorithm to insert a new node at the beginning

```
Step 1: IF AVAIL = NULL  
        Write OVERFLOW  
        Go to Step 13  
    [END OF IF]  
Step 2: SET NEW_NODE = AVAIL  
Step 3: SET AVAIL = AVAIL -> NEXT  
Step 4: SET NEW_NODE -> DATA = VAL  
Step 5: SET PTR = START  
Step 6: Repeat Step 7 while PTR -> NEXT != START  
Step 7:     SET PTR = PTR -> NEXT  
    [END OF LOOP]  
Step 8: SET PTR -> NEXT = NEW_NODE  
Step 9: SET NEW_NODE -> PREV = PTR  
Step 10: SET NEW_NODE -> NEXT = START  
Step 11: SET START -> PREV = NEW_NODE  
Step 12: SET START = NEW_NODE  
Step 13: EXIT
```

Algorithm to insert a new node at the end

```
Step 1: IF AVAIL = NULL
        Write OVERFLOW
        Go to Step 12
    [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:     SET PTR = PTR -> NEXT
    [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: SET START -> PREV = NEW_NODE
Step 12: EXIT
```

Algorithm to delete the first node

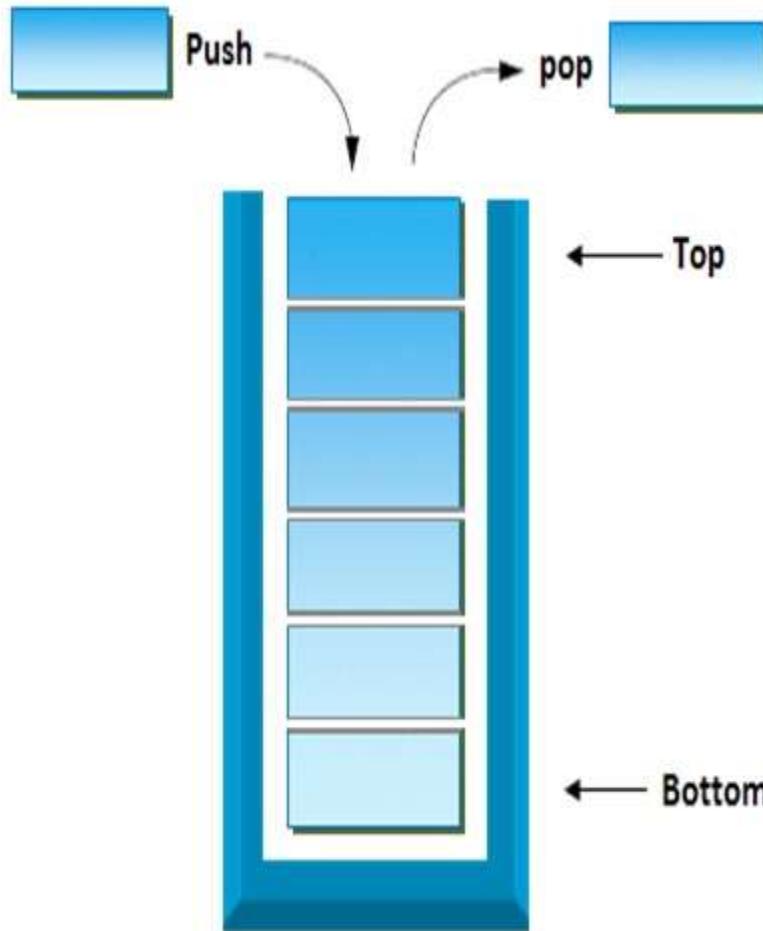
```
Step 1: IF START = NULL  
        Write UNDERFLOW  
        Go to Step 8  
    [END OF IF]  
Step 2: SET PTR = START  
Step 3: Repeat Step 4 while PTR → NEXT != START  
Step 4:     SET PTR = PTR → NEXT  
    [END OF LOOP]  
Step 5: SET PTR → NEXT = START → NEXT  
Step 6: SET START → NEXT → PREV = PTR  
Step 7: FREE START  
Step 8: SET START = PTR → NEXT
```

Algorithm to delete the last node

```
Step 1: IF START = NULL  
        Write UNDERFLOW  
        Go to Step 8  
    [END OF IF]  
Step 2: SET PTR = START  
Step 3: Repeat Step 4 while PTR -> NEXT != START  
Step 4:     SET PTR = PTR -> NEXT  
    [END OF LOOP]  
Step 5: SET PTR -> PREV -> NEXT = START  
Step 6: SET START -> PREV = PTR -> PREV  
Step 7: FREE PTR  
Step 8: EXIT
```

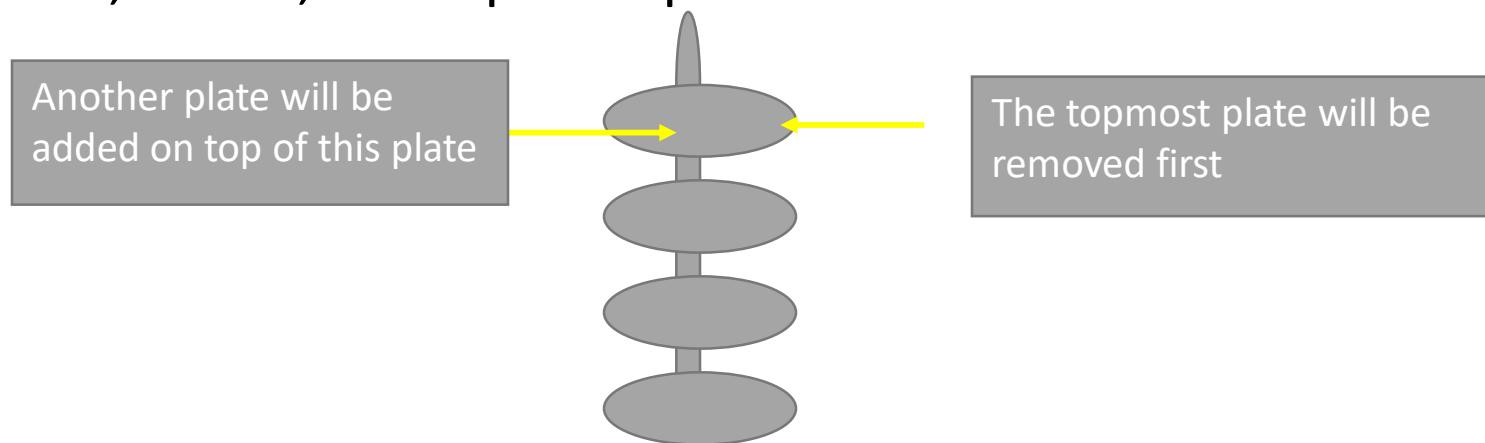
UNIT 2

Stacks

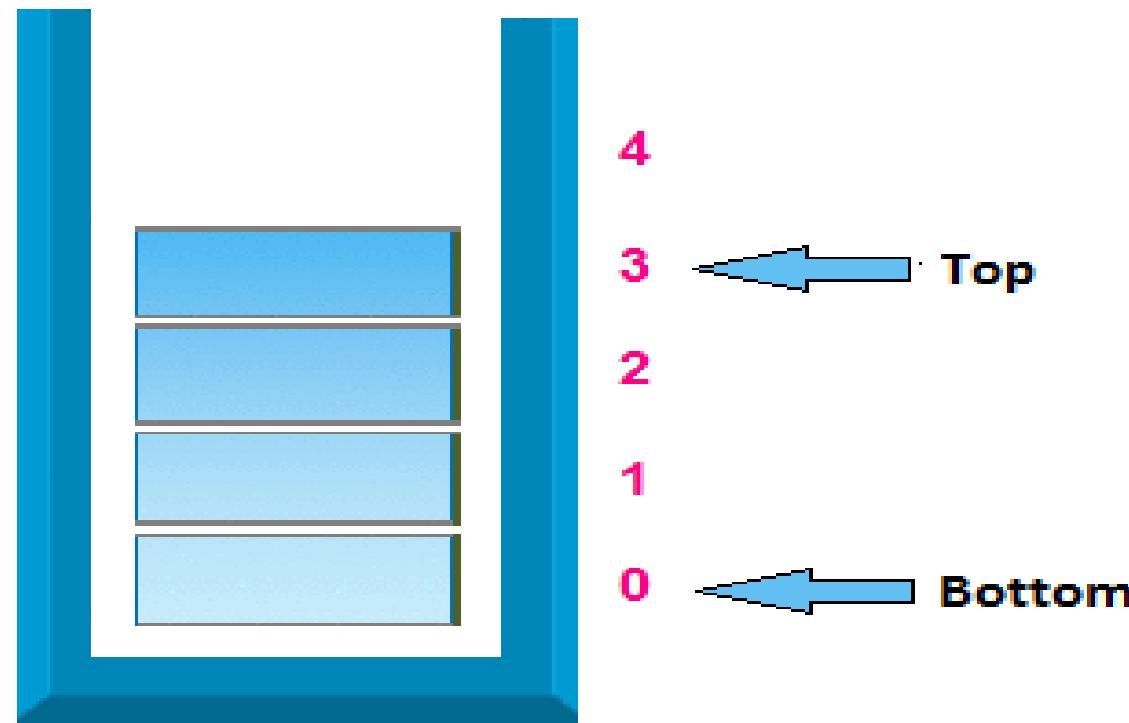


Introduction

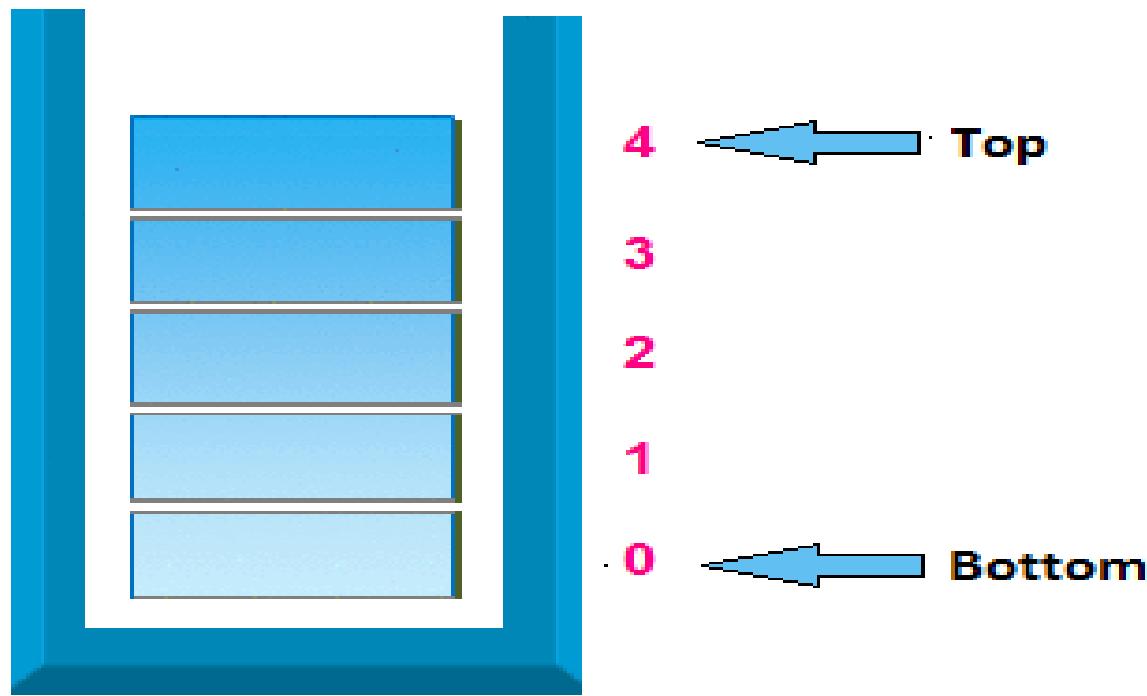
- Stack is an important data structure which stores its elements in an ordered manner.
- Take an analogy of a pile of plates where one plate is placed on top of the other. A plate can be removed only from the topmost position. Hence, you can add and remove the plate only at/from one position, that is, the topmost position.



PUSH OPERATION



POP OPERATION

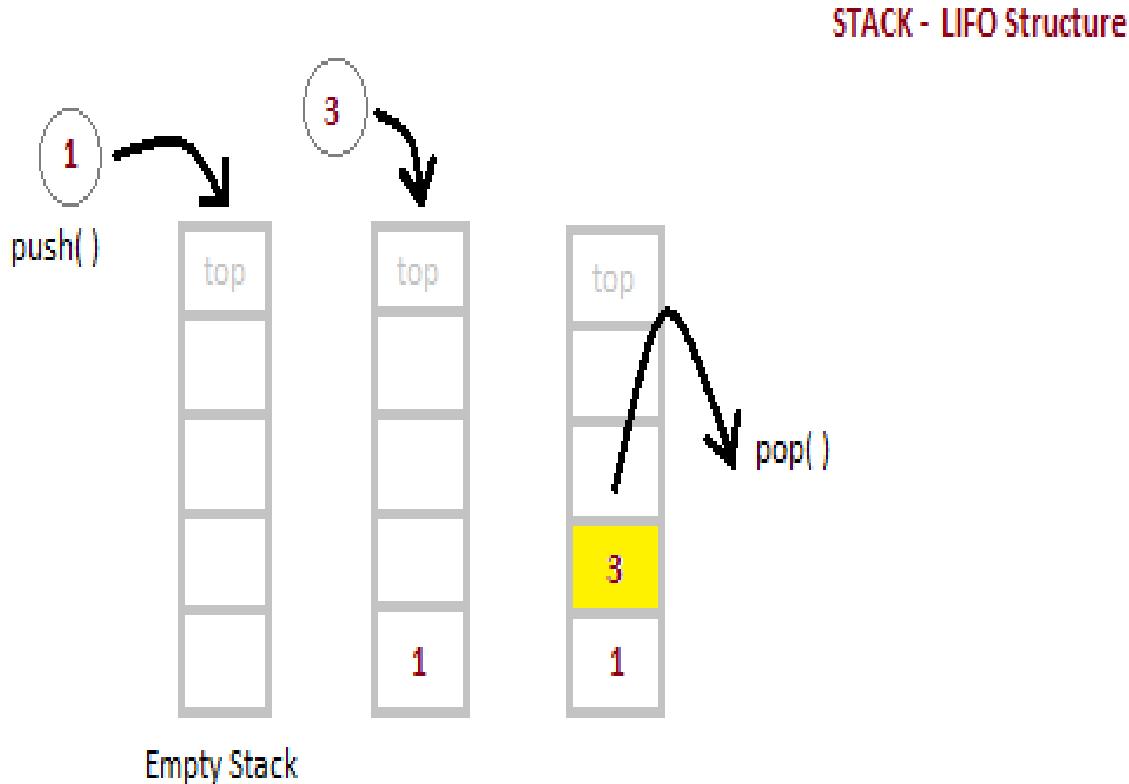


Stacks

- A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the *top*.
- Hence, a stack is called a LIFO (Last-In, First-Out) data structure as the element that is inserted last is the first one to be taken out.
- Stacks can be implemented either using an array or a linked list.

OPERATIONS OF STACKS

- PUSH
- POP
- PEEK/PEEP



In a Stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the Stack.

The "pop" operation removes the item on top of the stack.

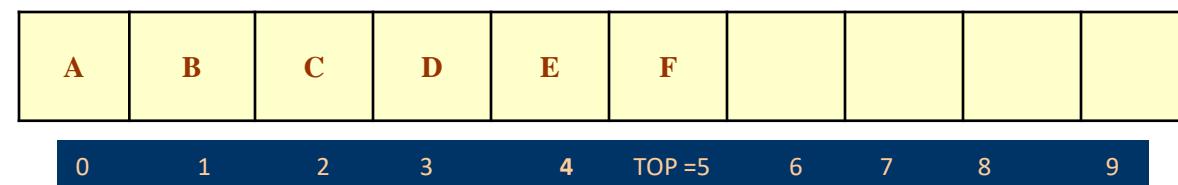
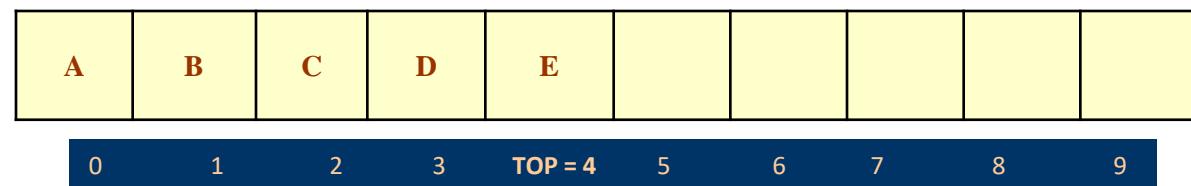
Array Representation of Stacks

Array Representation of Stacks

- In computer's memory stacks can be represented as a linear array.
- Every stack has a variable TOP associated with it.
- TOP is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted.
- There is another variable MAX which will be used to store the maximum number of elements that the stack can hold.
- If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX -1, then the stack is full.

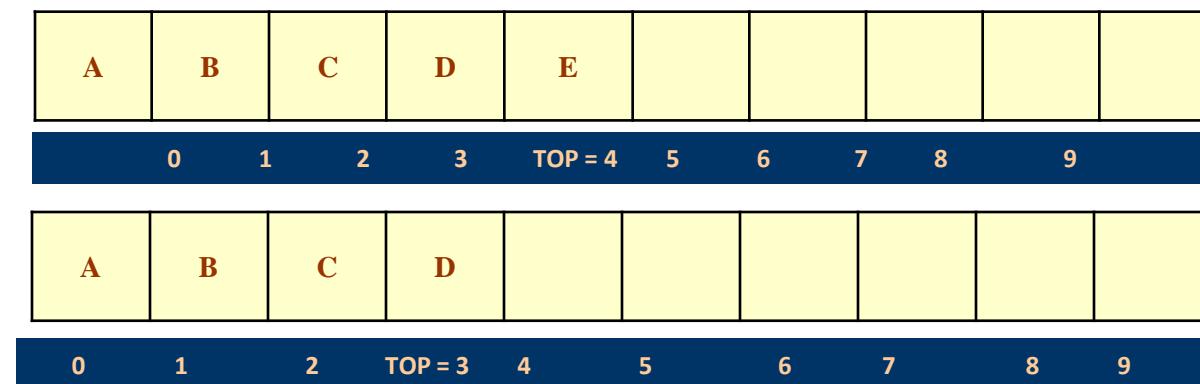
Push Operation

- The push operation is used to insert an element in to the stack.
- The new element is added at the topmost position of the stack.
- However, before inserting the value, we must first check if $\text{TOP}=\text{MAX}-1$, because if this is the case then it means the stack is full and no more insertions can further be done.
- If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.



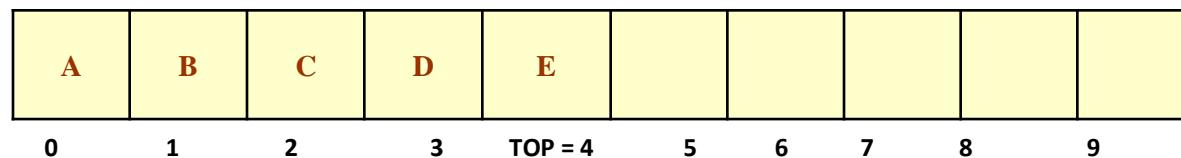
Pop Operation

- The pop operation is used to delete the topmost element from the stack.
- However, before deleting the value, we must first check if TOP=NULL , because if this is the case then it means the stack is empty so no more deletions can further be done.
- If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.



Peek Operation

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- However, the peek operation first checks if the stack is empty or contains some elements.
- If TOP = NULL, then an appropriate message is printed else the value is returned.



Here Peep operation will return E, as it is the value of the topmost element of the stack.

Algorithms for PUSH Operations

Step 1: IF TOP = MAX-1, then

PRINT "OVERFLOW"

Goto Step 4

[END OF IF]

Step 2: SET TOP = TOP + 1

Step 3: SET STACK[TOP] = VALUE

Step 4: END

Algorithms for POP Operations

Step 1: IF TOP = NULL, then

 PRINT "UNDERFLOW"

 Goto Step 4

 [END OF IF]

Step 2: SET STACK[TOP] = VALUE

Step 3: SET TOP = TOP - 1

Step 4: END

Algorithm for Peep Operation

Step 1: IF TOP =NULL, then

PRINT “STACK IS EMPTY”

Go TO Step 3

[END OF IF]

Step 2: RETURN STACK[TOP]

Step 3: END

LINKED LIST

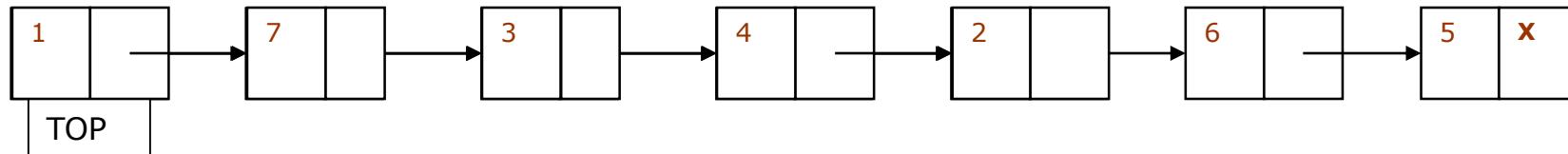
Representation

of

Stacks

Linear Representation of Stacks

- In a linked stack, every node has two parts – one that stores data and another that stores the address of the next node.
- The START pointer of the linked list is used as TOP.
- If TOP is NULL then it indicates that the stack is empty.



Push Operation on a Linked Stack

Step 1: Allocate memory for the new node and name it as `New_Node`

Step 2: SET `New_Node->DATA = VAL`

Step 3: IF `TOP = NULL`, then

 SET `New_Node->NEXT = NULL`

 SET `TOP = New_Node`

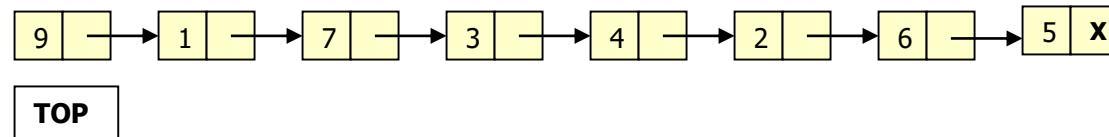
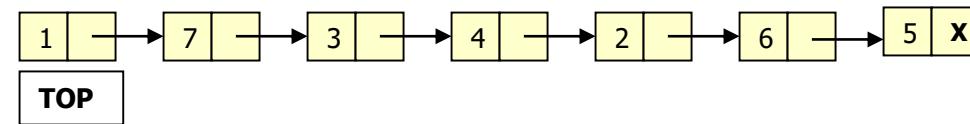
ELSE

 SET `New_node->NEXT = TOP`

 SET `TOP = New_Node`

[END OF IF]

Step 4: END



Pop Operation on a Linked Stack

Step 1: IF TOP = NULL, then

PRINT "UNDERFLOW"

Goto Step 5

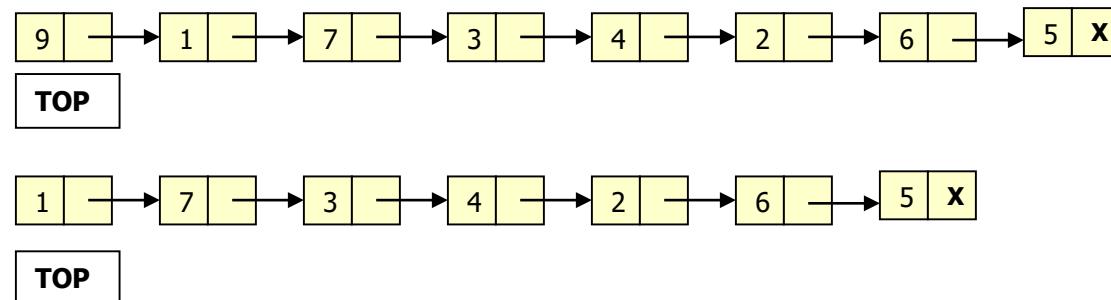
[END OF IF]

Step 2: SET PTR = TOP

Step 3: SET TOP = TOP ->NEXT

Step 4: FREE PTR

Step 5: END



Applications of Stacks

- Reversing a list
- Parentheses Checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Honai

Infix Notation

- Infix, Postfix and Prefix notations are three different but equivalent notations of writing algebraic expressions.
- While writing an arithmetic expression using infix notation, the operator is placed between the operands. For example, $A+B$; here, plus operator is placed between the two operands A and B.
- Although it is easy to write expressions using infix notation, computers find it difficult to parse as they need a lot of information to evaluate the expression.
- Information is needed about operator precedence, associativity rules, and brackets which overrides these rules.
- So, computers work more efficiently with expressions written using prefix and postfix notations.

Postfix Notation

- Postfix notation was given by Jan Lukasiewicz who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation which is better known as Reverse Polish Notation or RPN.
- In postfix notation, the operator is placed after the operands. For example, if an expression is written as $A+B$ in infix notation, the same expression can be written as $AB+$ in postfix notation.
- The order of evaluation of a postfix expression is always from left to right.

Postfix Notation

- The expression $(A + B) * C$ is written as:
 $AB+C*$ in the postfix notation.
- A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands.
- For example, given a postfix notation $AB+C^*$. While evaluation, addition will be performed prior to multiplication.

Prefix Notation

- In a prefix notation, the operator is placed before the operands.
- For example, if $A+B$ is an expression in infix notation, then the corresponding expression in prefix notation is given by $+AB$.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
- Prefix expressions also do not follow the rules of operator precedence, associativity, and even brackets cannot alter the order of evaluation.
- The expression $(A + B) * C$ is written as:
 $*+ABC$ in the prefix notation

Evaluation of an Infix Expression to Postfix

Algorithm to convert an Infix notation into postfix notation

Step 1: Add ‘)’ to the end of the infix expression

Step 2: Push “(“ on to the stack

Step 3: Repeat until each character in the infix notation is scanned

 IF a “(“ is encountered, push it on the stack

 IF an operand (whether a digit or an alphabet) is encountered,
 add it to the postfix expression.

 IF a “)” is encountered, then;

 A. Repeatedly pop from stack and add it to the postfix expression until a
 “(“ is encountered.

 B. Discard the “(“. That is, remove the “(“ from stack and do not add it
 to the postfix expression

 IF an operator X is encountered, then;

 Repeatedly pop from stack and add each operator (popped from the
 stack) to the postfix expression which has the same precedence or a
 higher precedence than X

 b. Push the operator X to the stack

Step 4: Repeatedly pop from the stack and add it to the postfix expression

Infix Notation to Post fix Notation

Step 1: Add ‘)’ to the end of the infix expression

Step 2: Push “(“ on to the stack

Step 3: Repeat until each character in the infix notation is scanned

 IF a “(“ is encountered, push it on the stack

 IF an operand (whether a digit or an alphabet) is encountered,
 add it to the postfix expression.

 IF a “)” is encountered, then;

Repeatedly pop from stack and add it to the postfix expression until a “(“
is encountered.

Discard the “(“. That is, remove the “(“ from stack and do not add it to
the postfix expression

Evaluation of an Infix Expression

Step 1: Add a “)” at the end of the postfix expression

Step 2: Scan every character of the postfix expression and repeat
steps 3 and 4 until “)” is encountered

Step 3: IF an operand is encountered, push it on the stack

IF an operator X is encountered, then

- a. pop the top two elements from the stack as A and B
- b. Evaluate $B \times A$, where A was the topmost element and B was the element below A.
- c. Push the result of evaluation on the stack

[END OF IF]

Step 4: SET RESULT equal to the topmost element of the stack

Step 5: EXIT

Evaluation of an Infix Expression

- Let us now take an example that makes use of this algorithm.
- Consider the infix expression given as “9 - ((3 * 4) + 8) / 4”.
- The infix expression "9 - ((3 * 4) + 8) / 4" can be written as "9 3 4 * 8 + 4 / -“ using postfix notation.
- Look at table which shows the procedure.

Character scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Convert Infix Expression into Prefix Expression

Consider an infix expression: $(A - B / C) * (A / K - L)$

- Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parenthesis.
$$(L - K / A) * (C / B - A)$$
- Step 2: Obtain the corresponding postfix expression of the infix expression obtained as a result of Step 1.
- The expression is: $(L - K / A) * (C / B - A)$
- Therefore,
$$\begin{aligned} & [L - (K A /)] * [(C B /) - A] \\ &= [LKA/-] * [CB/A-] \\ &= L K A / - C B / A - * \end{aligned}$$
- Step 3: Reverse the postfix expression to get the prefix expression
- Therefore, the prefix expression is $* - A / B C - / A K L$

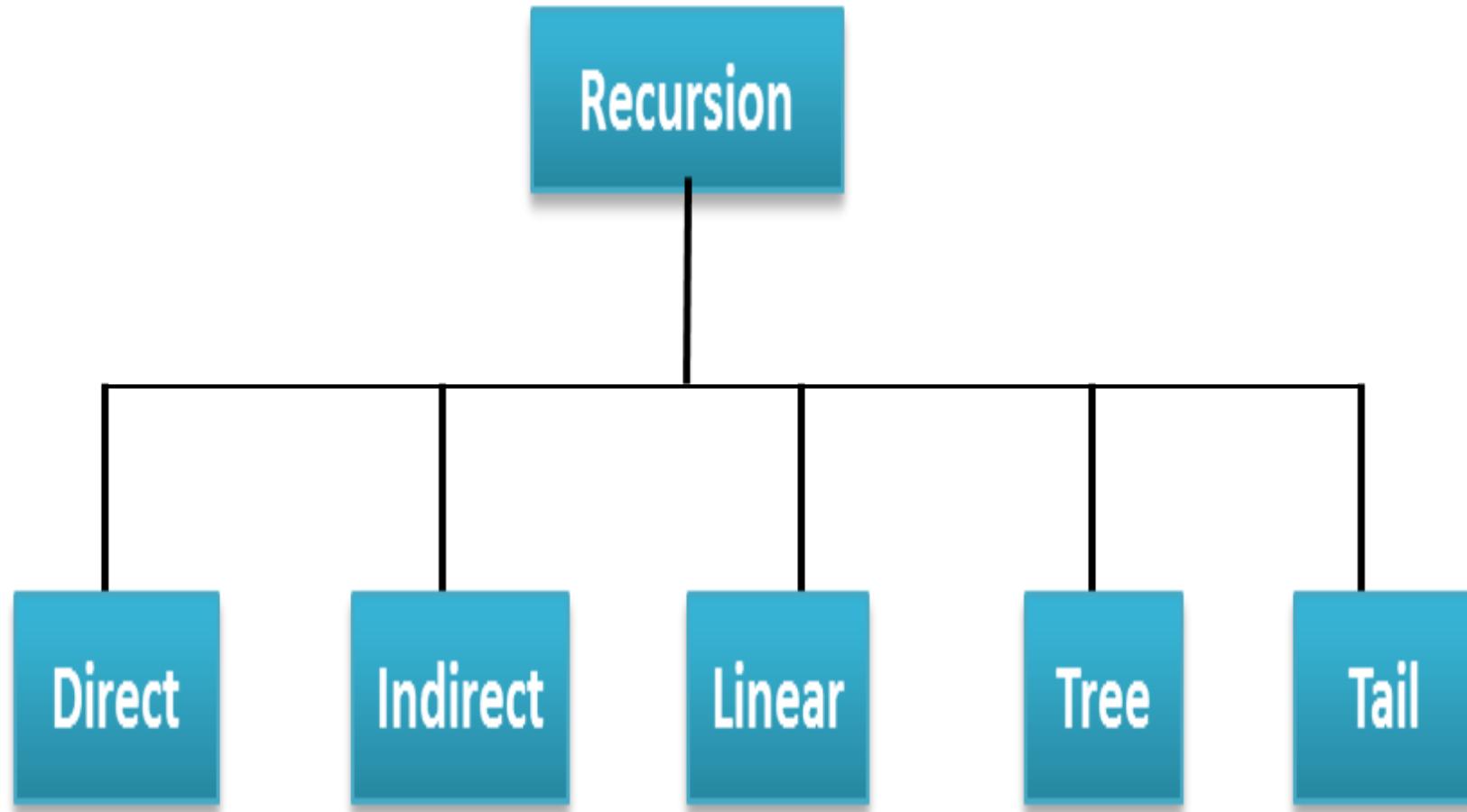
Recursion

Recursion

- Recursion is an implicit application of STACK ADT.
- A recursive function is a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Every recursive solution has two major cases: the *base case* in which the problem is simple enough to be solved directly without making any further calls to the same function.
- *Recursive case*, in which first the problem at hand is divided into simpler subparts. Second the function calls itself but with subparts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Types of Recursion

- Any recursive function can be characterized based on:
 - whether the function calls itself directly or indirectly (direct or indirect recursion).
 - whether any operation is pending at each recursive call (tail-recursive or not).
 - the structure of the calling pattern (linear or tree-recursive).



Direct Recursion

- A function is said to be *directly* recursive if it explicitly calls itself.
- For example, consider the function given below.

```
int Func( int n)
{
    if(n==0)
        retrun n;
    return (Func(n-1));
}
```

Indirect Recursion

- A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it.
- Look at the functions given below. These two functions are indirectly recursive as they both call each other.

```
int Func1(int n)
{
    if(n==0)
        return n;
    return Func2(n);
}
```

```
int Func2(int x)
{
    return Func1(x-
1);
}
```

Linear Recursion

- Recursive functions can also be characterized depending on the way in which the recursion grows: in a linear fashion or forming a tree structure.
- In simple words, a recursive function is said to be *linearly* recursive when no pending operation involves another recursive call to the function.
- For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to *fact()* function.

Tree Recursion

- A recursive function is said to be *tree* recursive (or *non-linearly* recursive) if the pending operation makes another recursive call to the function.
- For example, the Fibonacci function Fib in which the pending operations recursively calls the Fib function.

```
int Fibonacci(int num)
{
    if(num <= 2)
        return 1;
    return ( Fibonacci (num - 1) + Fibonacci(num - 2));
}
```

Tail Recursion

- A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller.
- That is, when the called function returns, the returned value is immediately returned from the calling function.
- Tail recursive functions are highly desirable because they are much more efficient to use as in their case, the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

```
int Fact(n)
{
    return
    Fact1(n, 1);
}
```

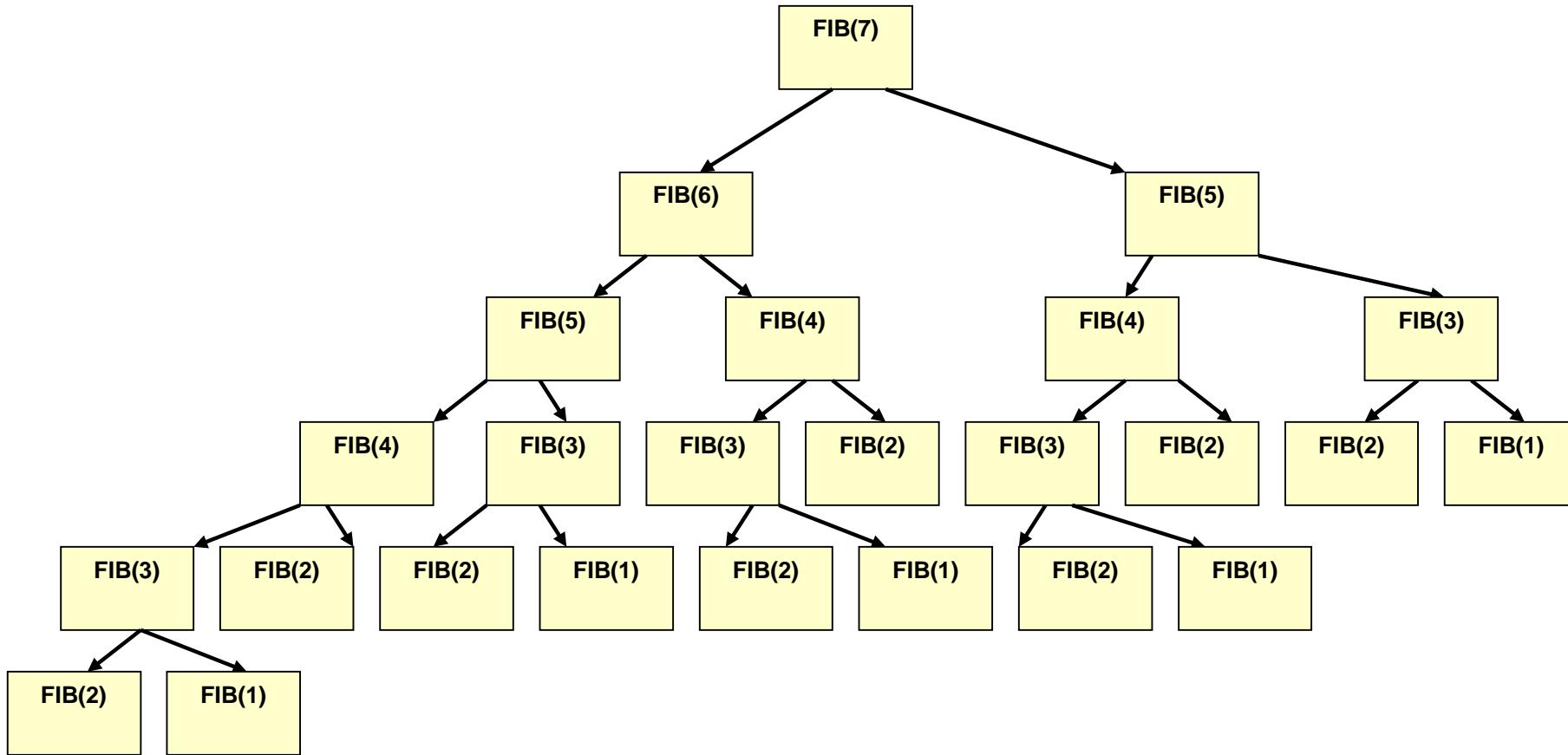
```
int Fact1(int n, int res)
{
    if (n==1)
        return res;
    return Fact1(n-1,
                n*res);
}
```

Fibonacci Series

- The Fibonacci series can be given as:
0 1 1 2 3 5 8 13 21 34 55.....
- That is, the third term of the series is the sum of the first and second terms. Similarly, fourth term is the sum of second and third terms, so on and so forth.
- A recursive solution to find the nth term of the Fibonacci series can be given as:

FIB(n) =1, if n<=2

FIB (n - 1) + FIB (n - 2), otherwise



Pros and Cons of Recursion

Pros

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

Cons

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive process in midstream is slow.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counterpart.
- It is difficult to find bugs, particularly when using global variables.

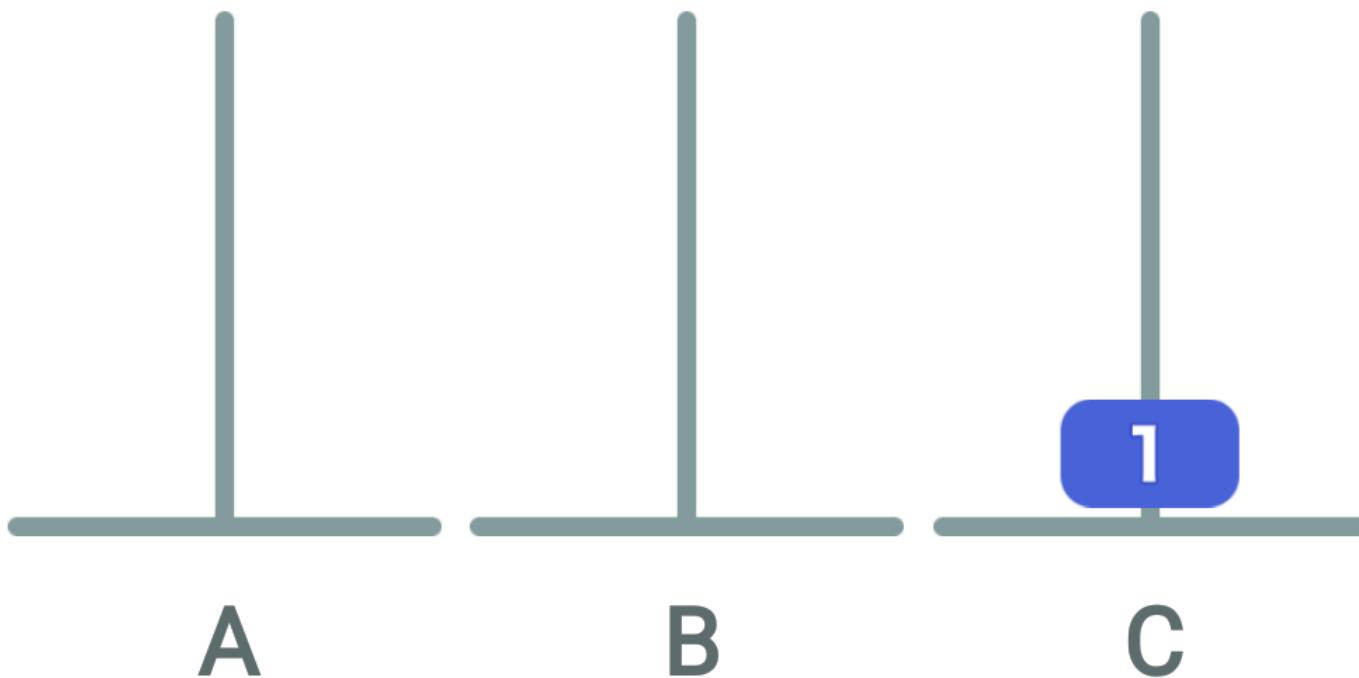
**TOWER
OF
HANOI**

ONE DISK

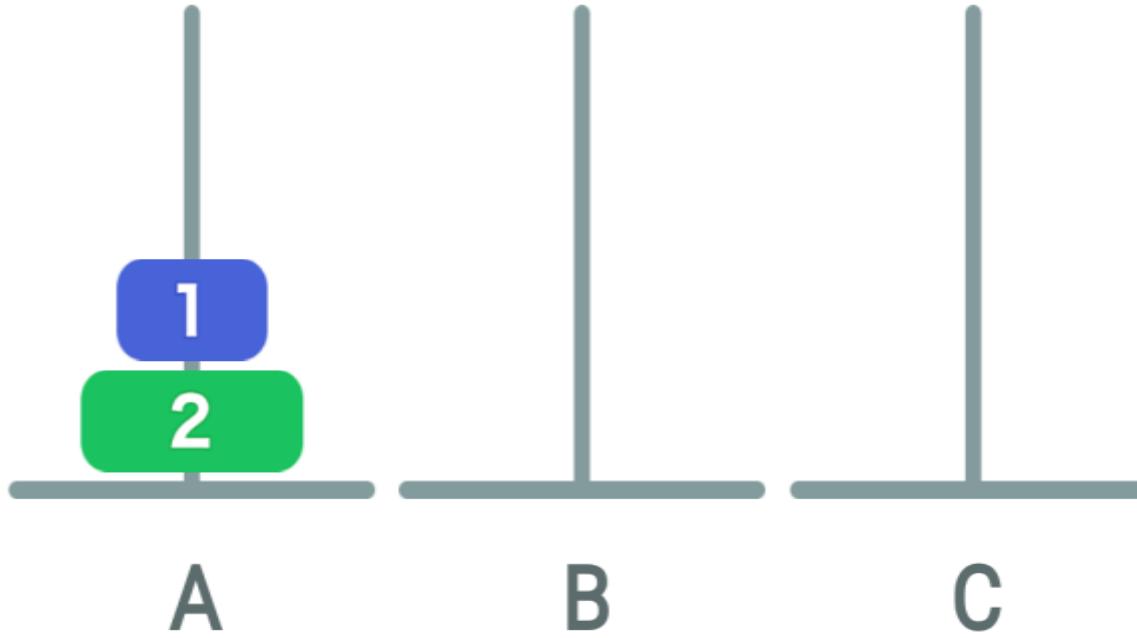


We were able to reach the goal when there was 1 disk.

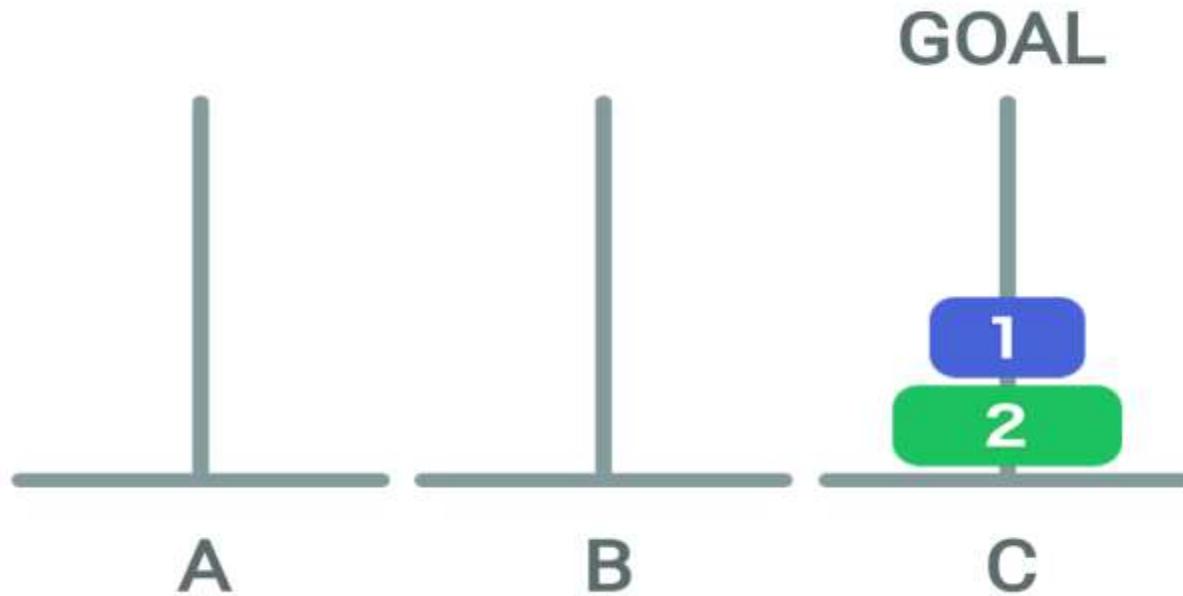
GOAL



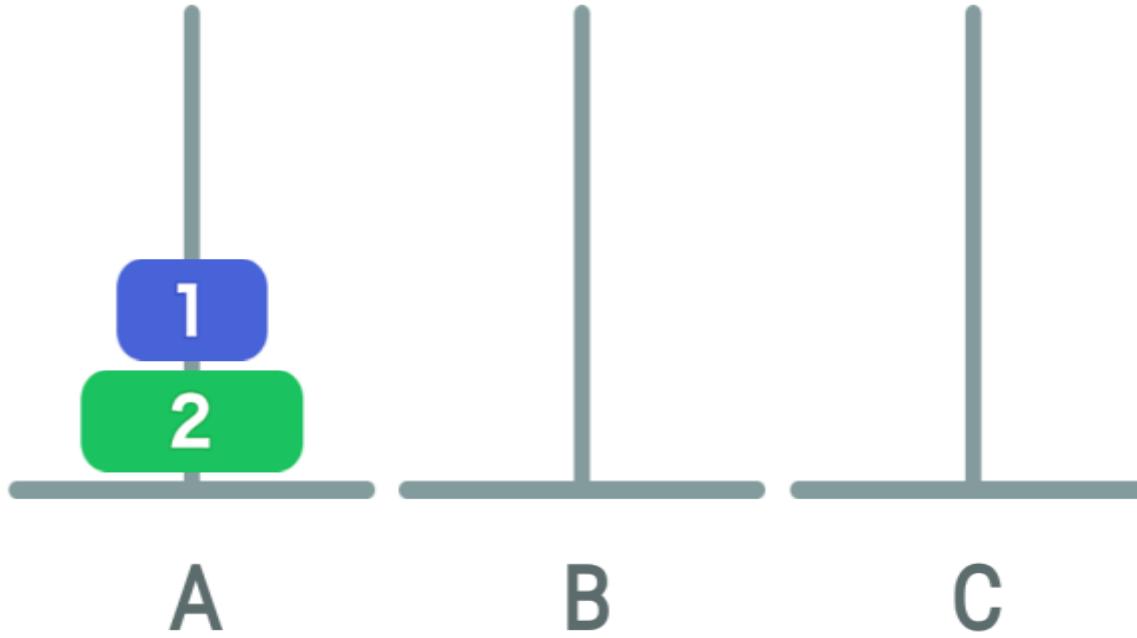
TWO DISK



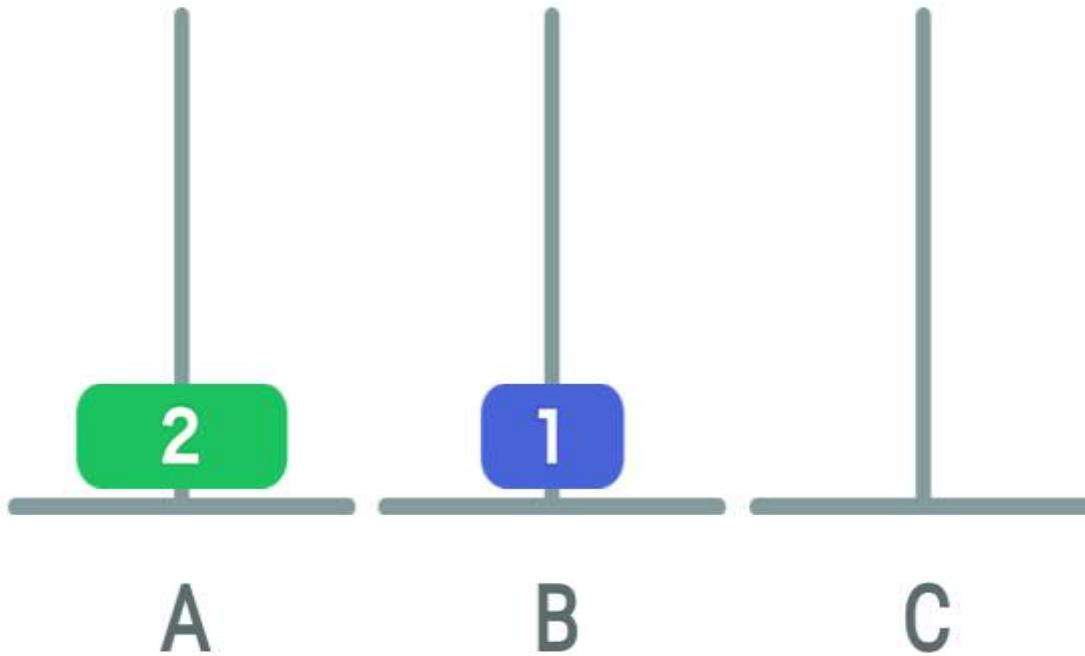
The diagram shows 3 stakes, A, B, and C, with two disks on stake A.



The goal is to move the disks onto stake C while keeping the disks in the same order as stake A.

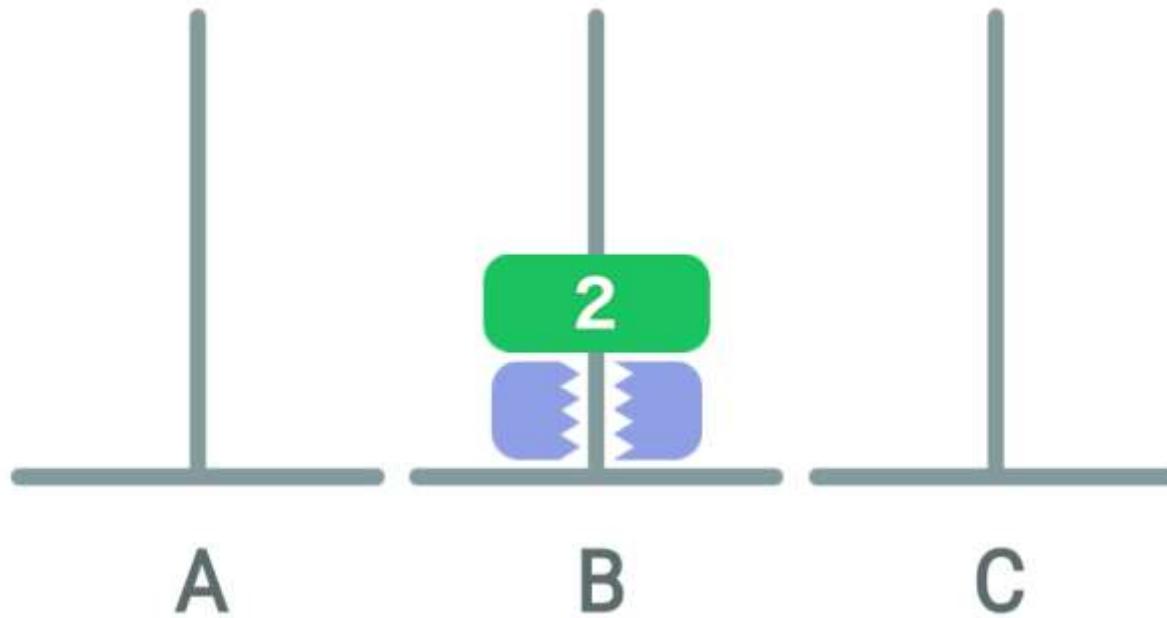


There are two conditions for moving disks.
The first condition is that you can only
move one disk at a time.

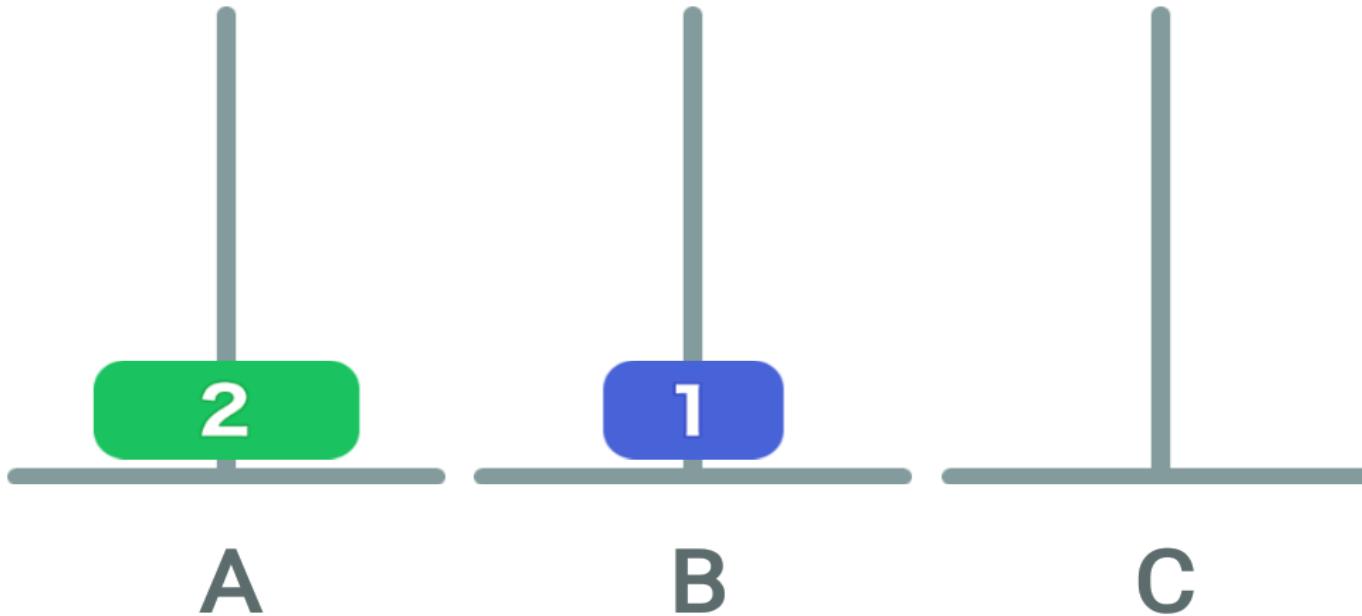


Moving 1 disk like this is no problem...

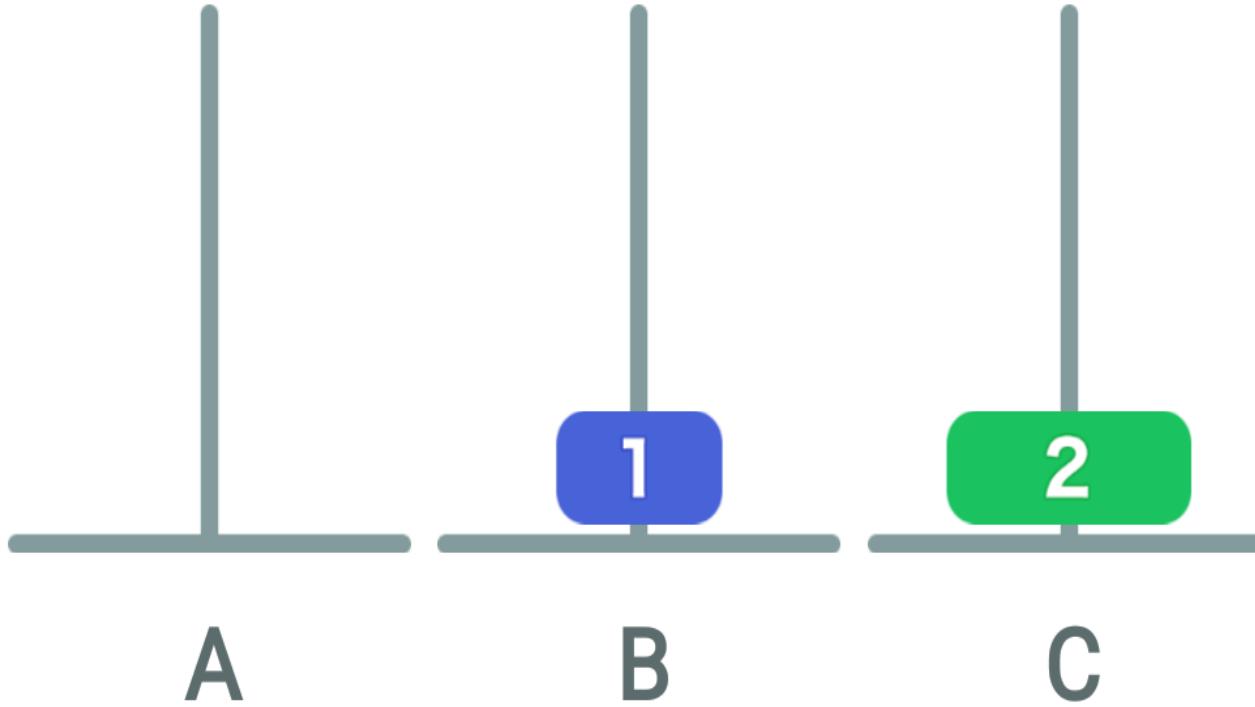
The 2nd condition is that you can't set a bigger disk on top of a smaller disk.



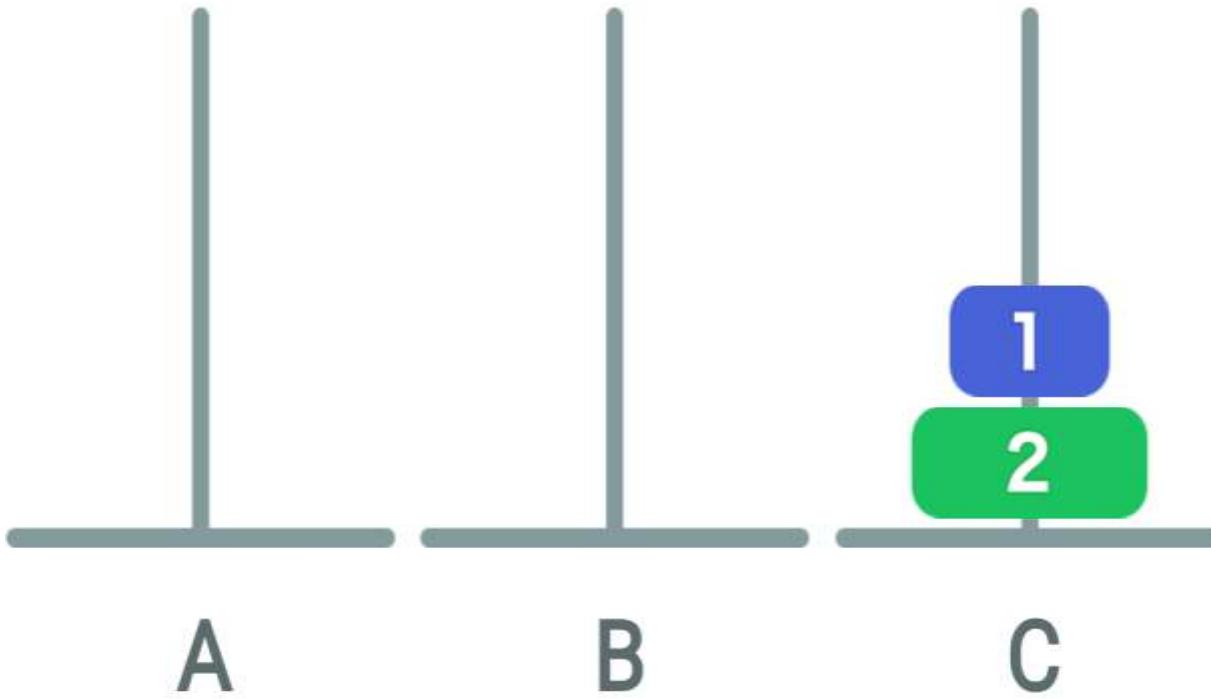
The 2nd condition is that you can't set a bigger disk on top of a smaller disk.



The smaller disk is on top, so we can move it to stake B.



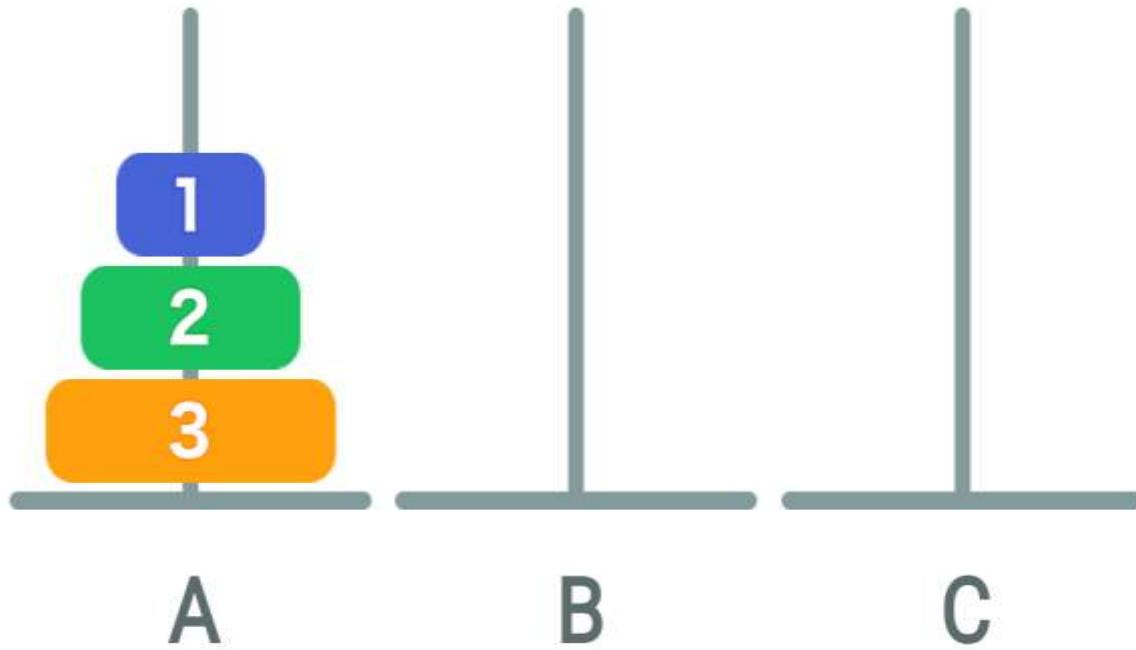
We move the bigger disk to stake C.



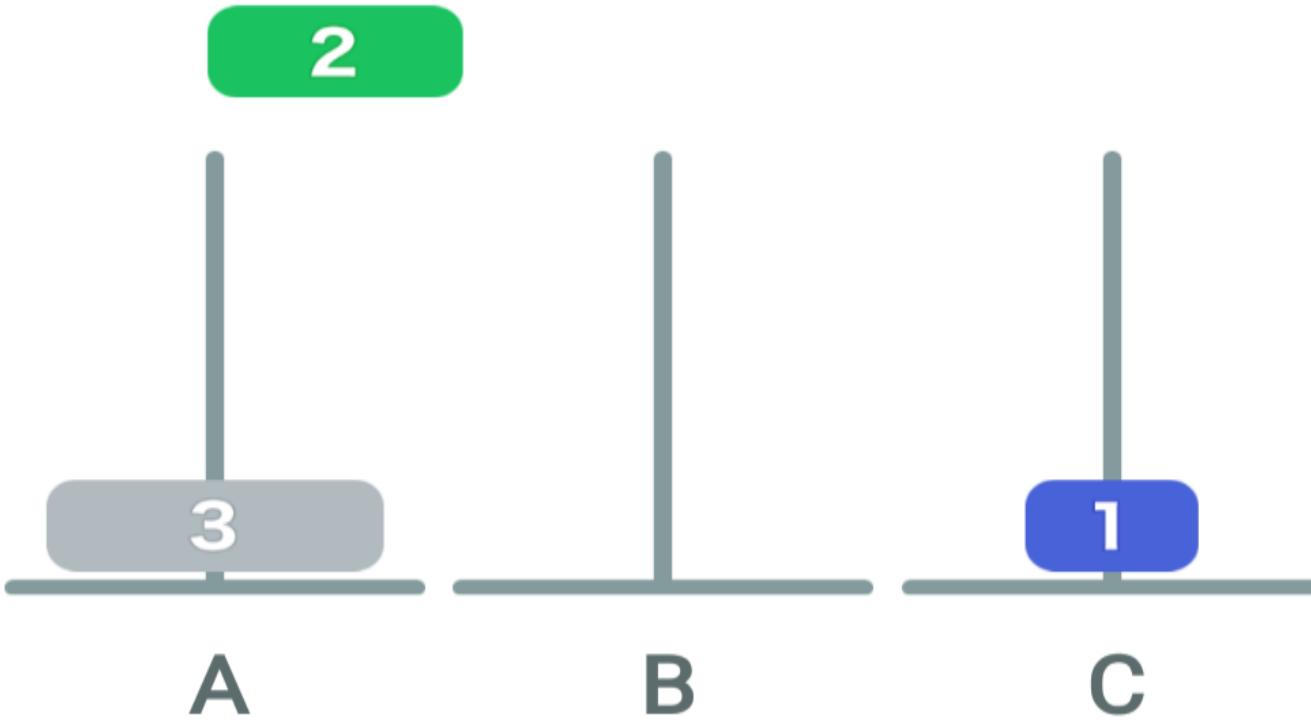
We move the bigger disk to stake C.

And the Goal is Reached

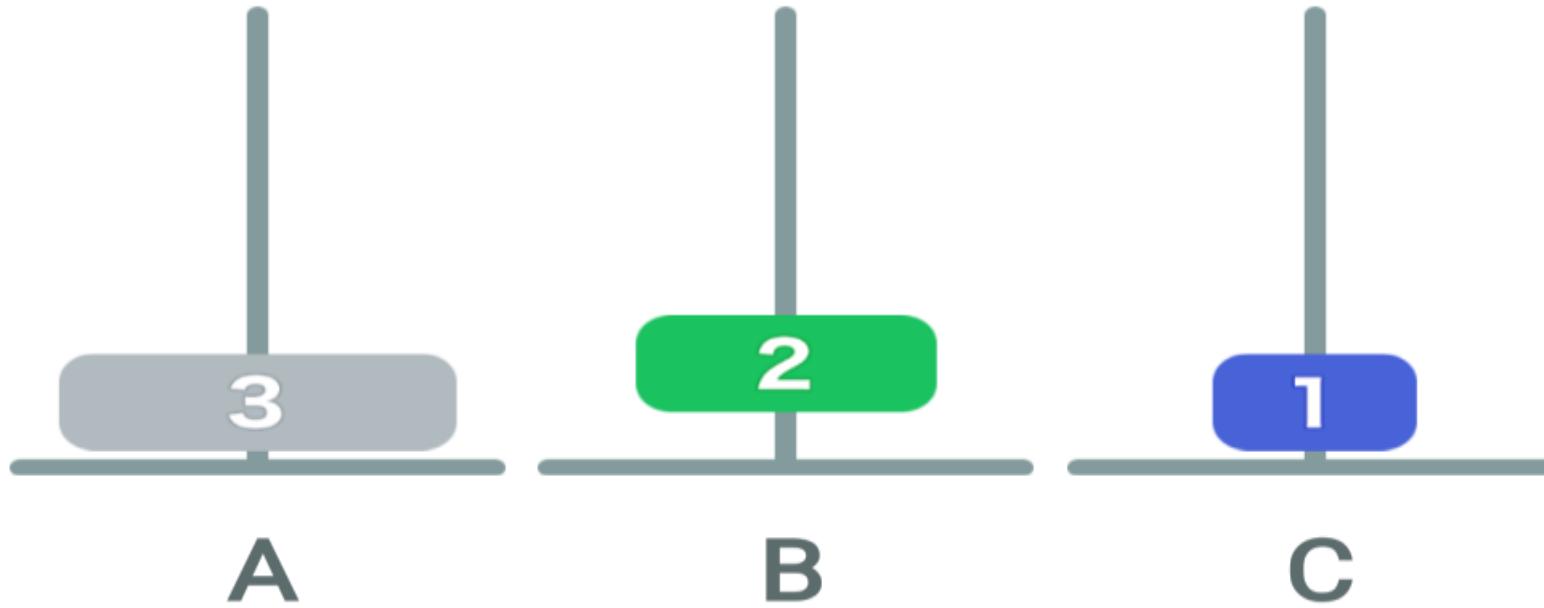
3 DISKS



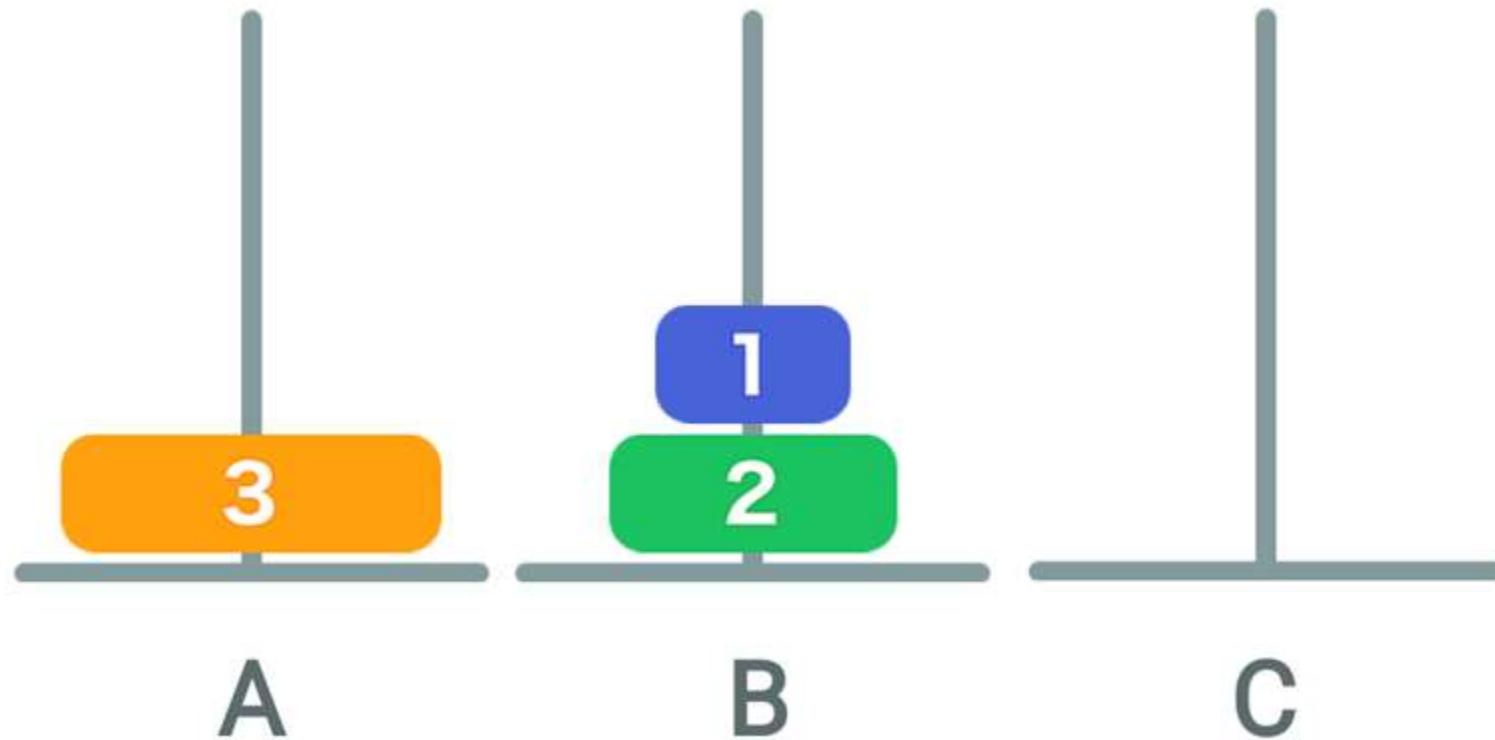
What about when there are 3 disks?

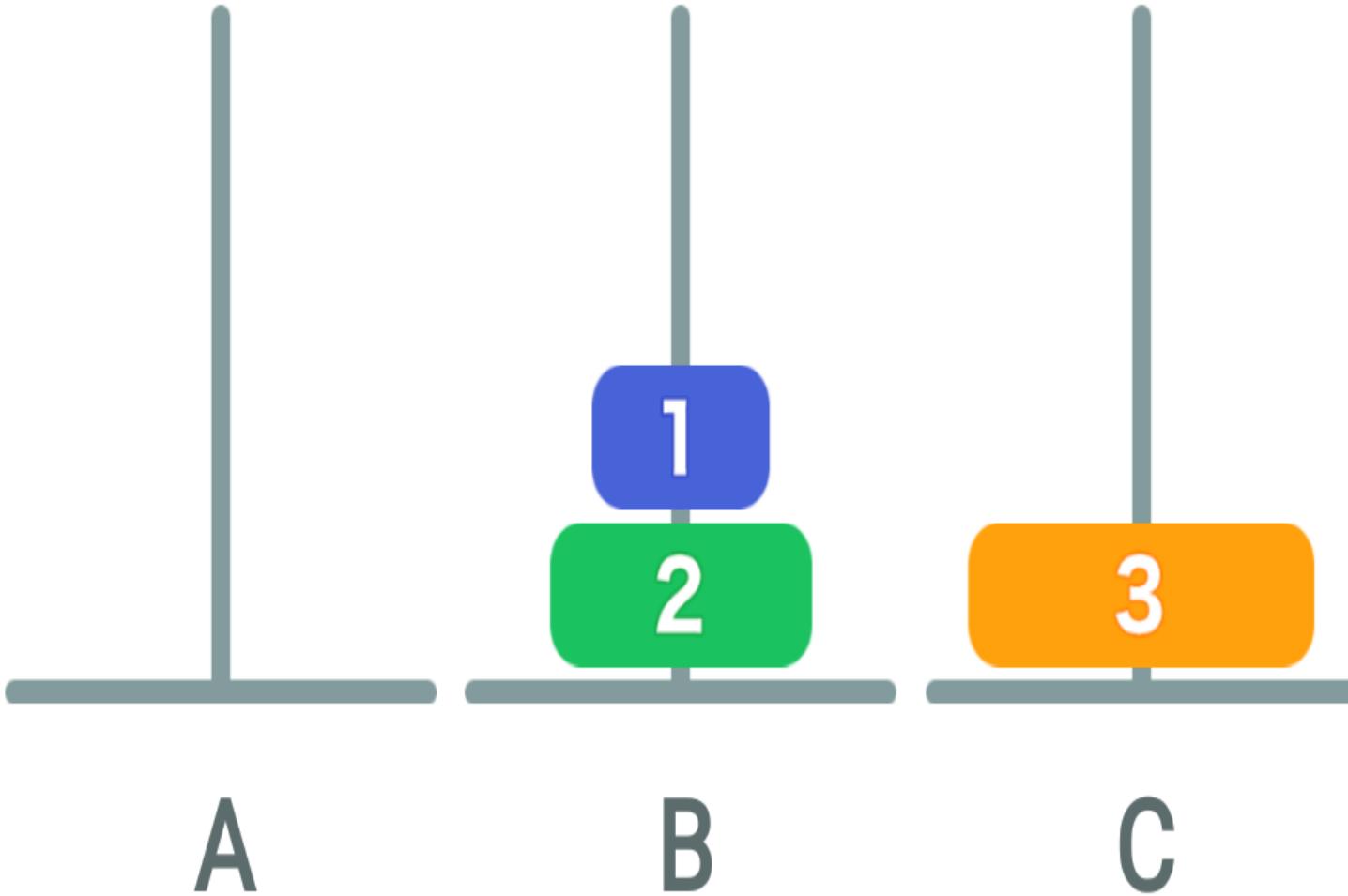


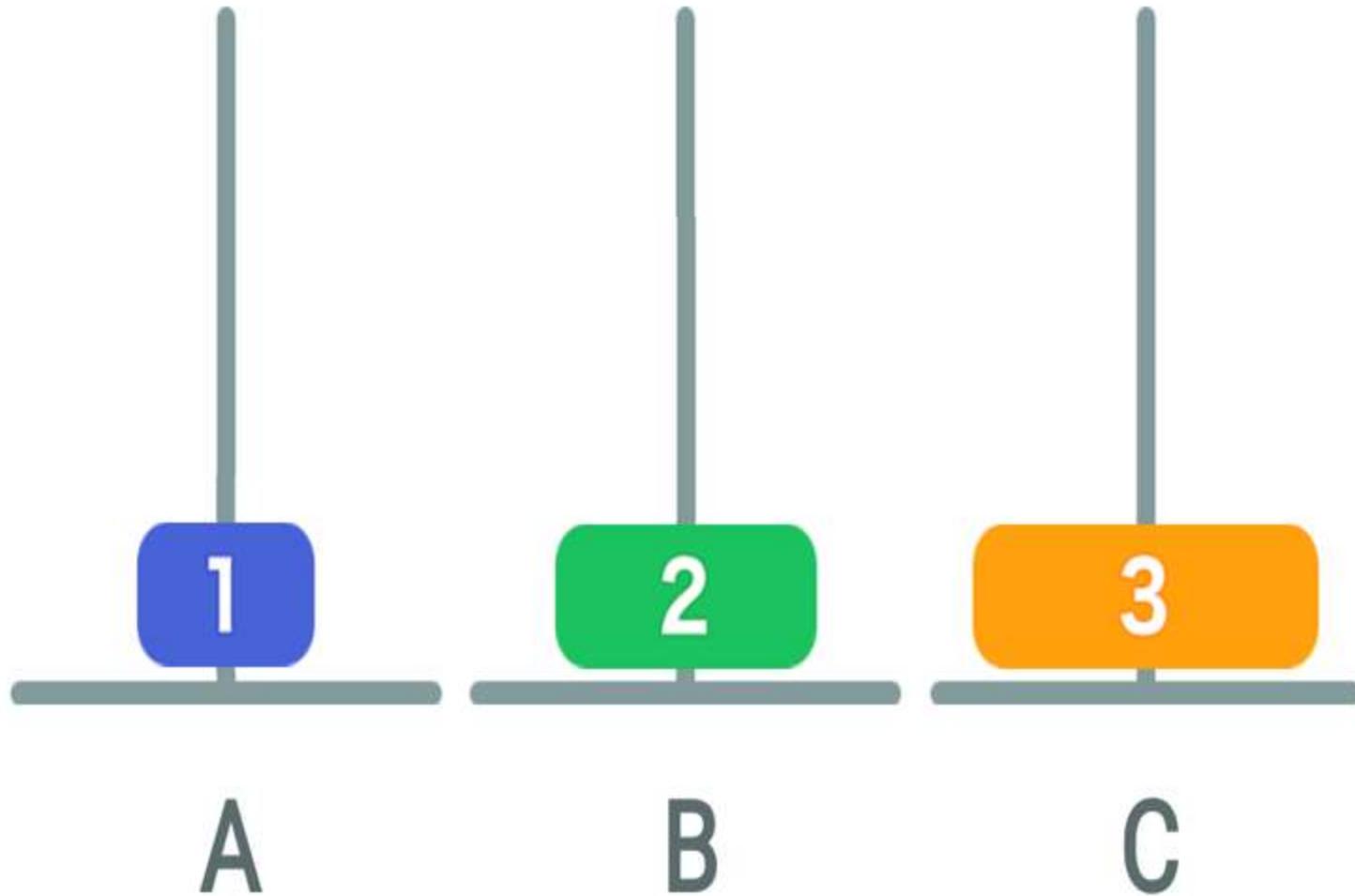
If we move the remaining disks in the same general way as before when there were only 2 disks, we're able to move them to stake B.

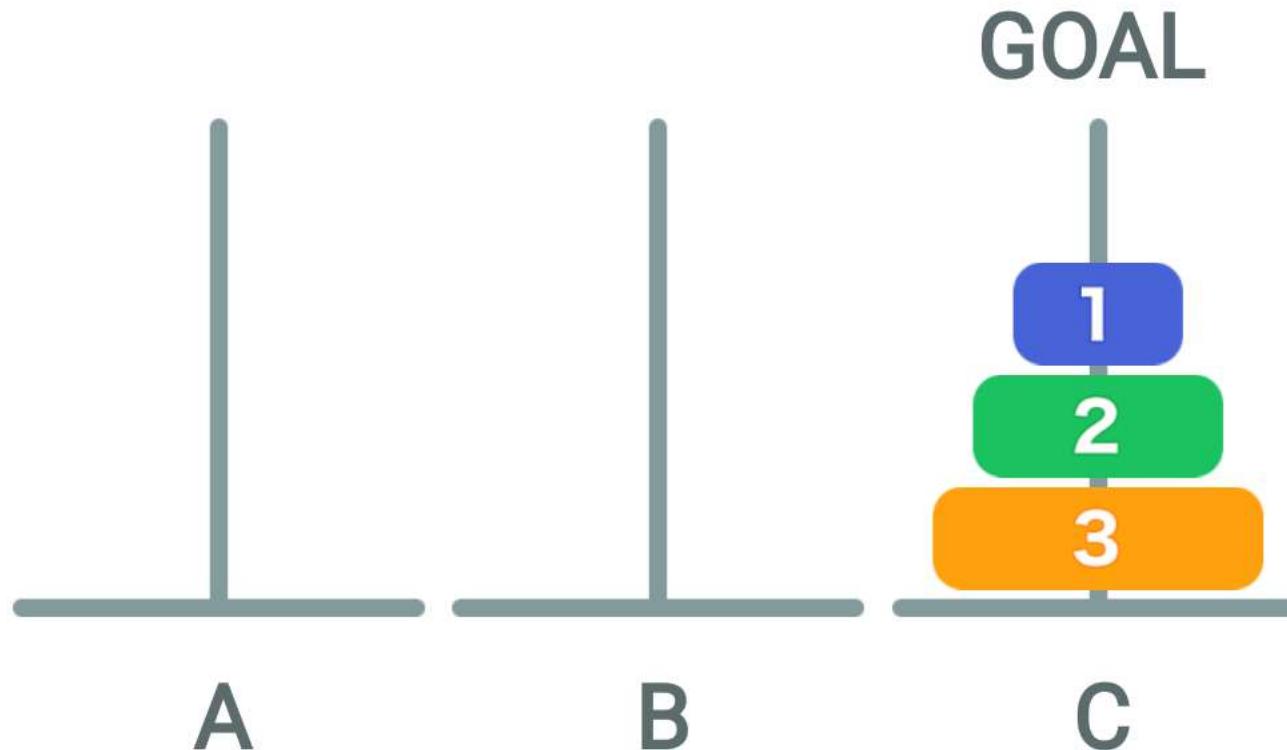


If we move the remaining disks in the same general way as before when there were only 2 disks, we're able to move them to stake B.





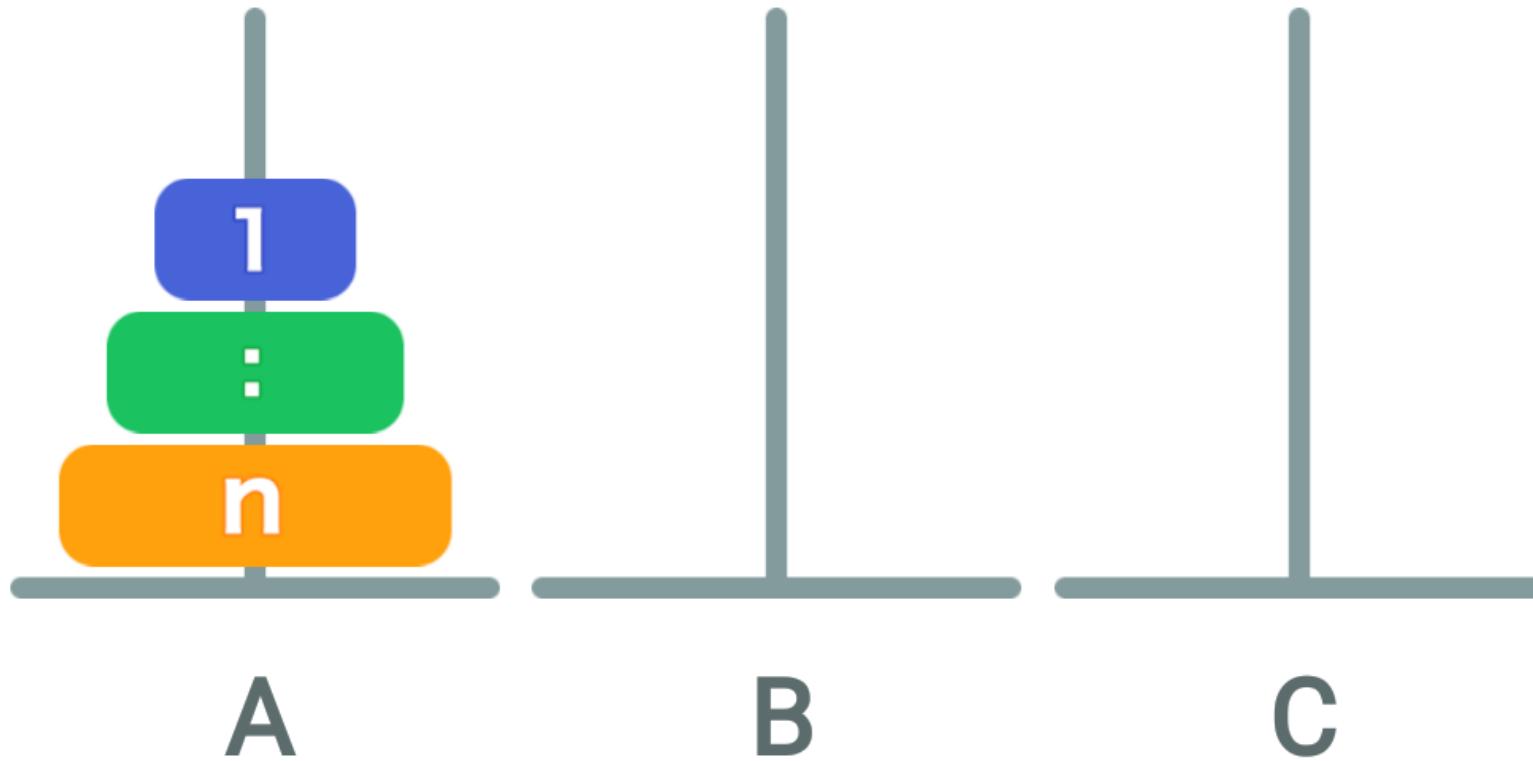




Then, using the same principles as before,
we move the disks on stake B to stake C.

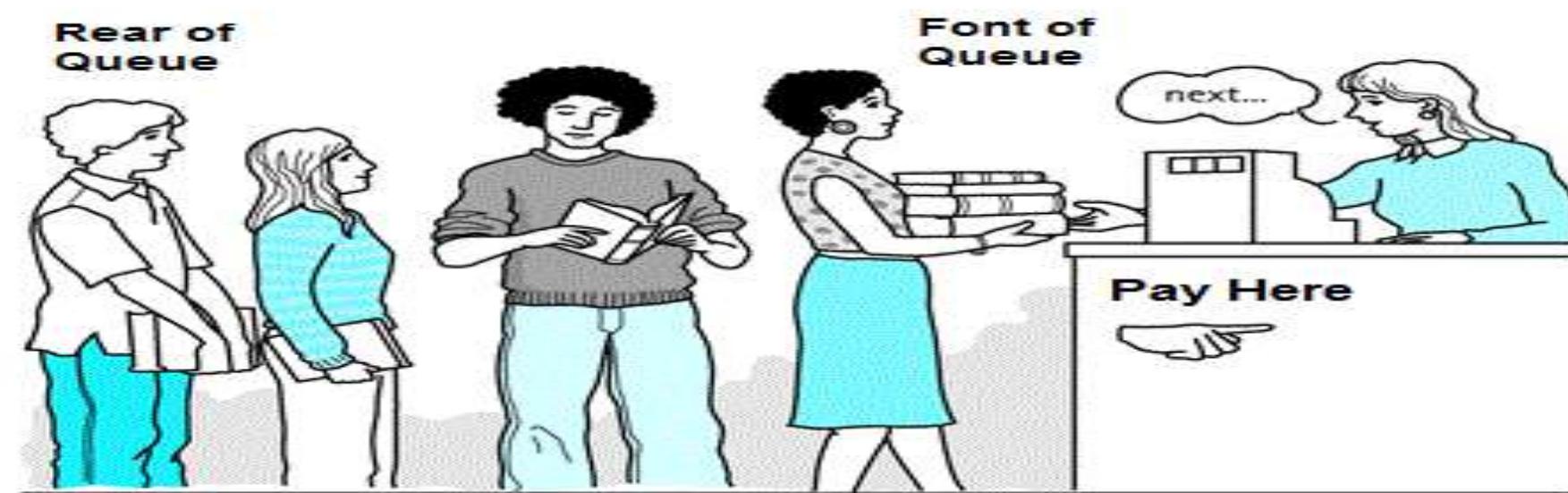
And the Goal is Reached

N DISKS



Let's assume that we can reach the goal when there are n disks.

QUEUES

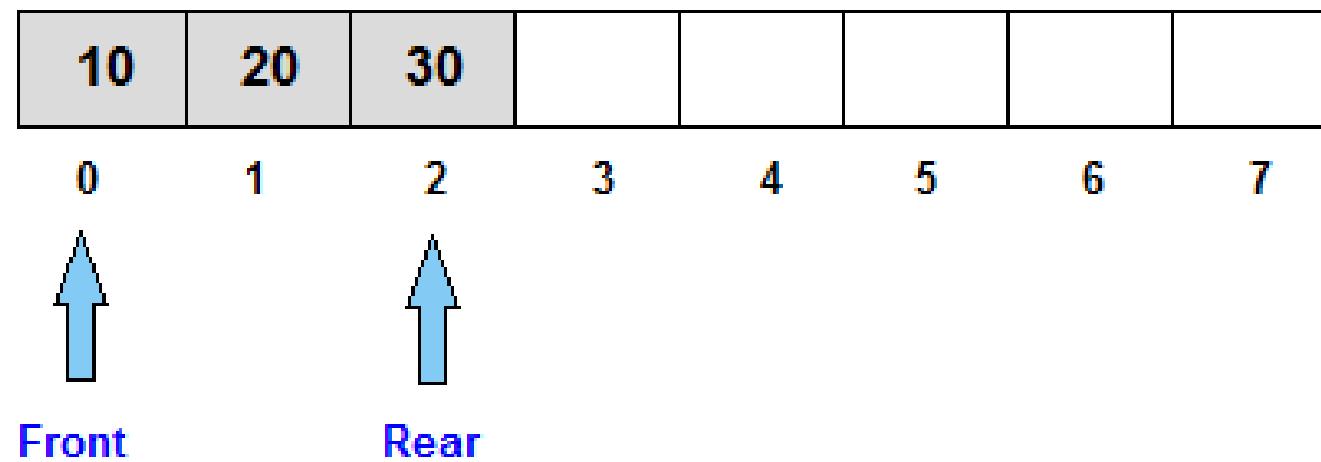


Real Life Example of Queue : Library Counter

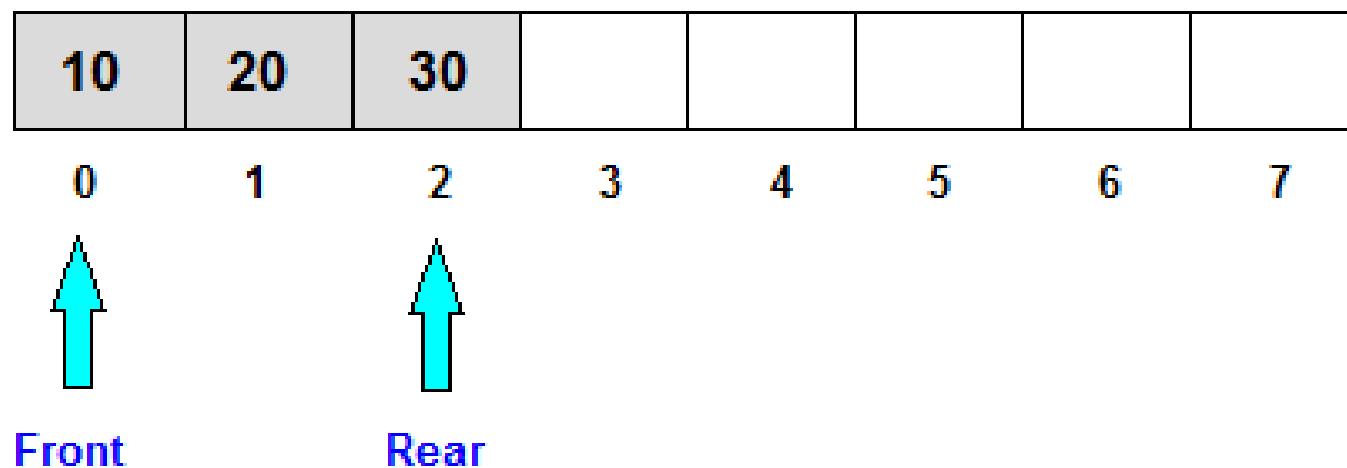
Introduction

- Queue is an important data structure which stores its elements in an ordered manner.
- We can explain the concept of queues using the following analogy:
People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the *rear* and removed from the other one end called the *front*.

INSERTING AN ELEMENT



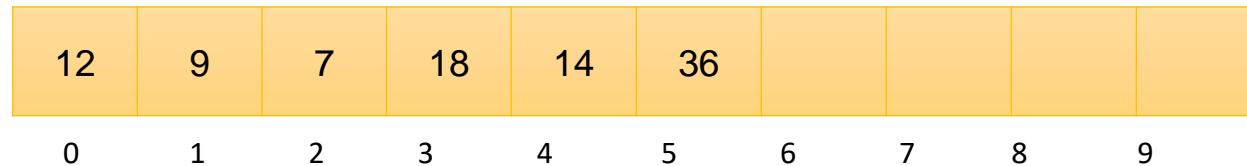
DELETING A ELEMENT



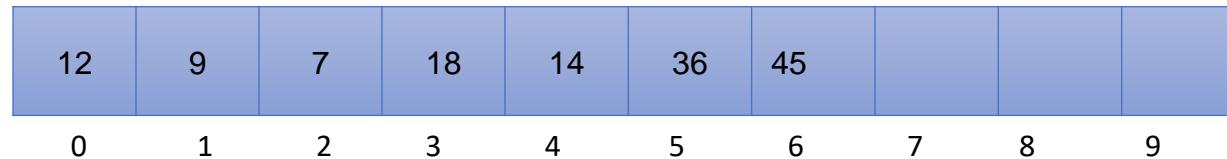
Array Representation of Queues

Array Representation of Queues

- Queues can be easily represented using linear arrays.
- Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.
- Consider the queue shown in figure

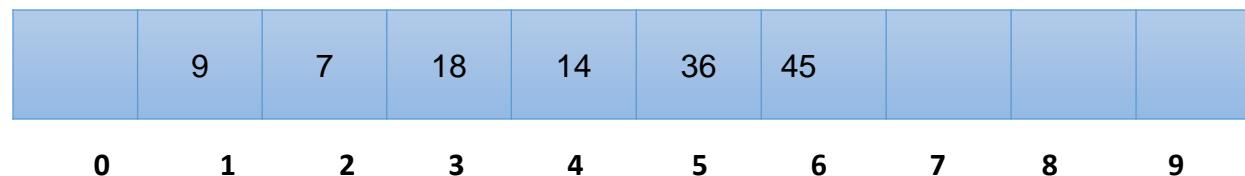


- Here, front = 0 and rear = 5.
- If we want to add one more value in the list say with value 45, then rear would be incremented by 1 and the value would be stored at the position pointed by rear.



Array Representation of Queues

- Now, front = 0 and rear = 6. Every time a new element has to be added, we will repeat the same procedure.
- Now, if we want to delete an element from the queue, then the value of front will be incremented. Deletions are done from only this end of the queue.



- Now, front = 1 and rear = 6.

Array Representation of Queues

- Before inserting an element in the queue we must check for overflow conditions.
- An overflow occurs when we try to insert an element into a queue that is already full, i.e. when $\text{rear} = \text{MAX} - 1$, where MAX specifies the maximum number of elements that the queue can hold.
- Similarly, before deleting an element from the queue, we must check for underflow condition.
- An underflow occurs when we try to delete an element from a queue that is already empty. If $\text{front} = -1$ and $\text{rear} = -1$, this means there is no element in the queue.

Algorithm for Insertion Operation

Step 1: IF REAR=MAX-1, then;

Write OVERFLOW

Go to Step 4

[END OF IF]

Step 2: IF FRONT == -1 and REAR = -1, then

SET FRONT = REAR = 0

ELSE

SET REAR = REAR + 1

[END OF IF]

Step 3: SET QUEUE[REAR] = NUM

Step 4: Exit

Algorithm for Deletion Operation

Step 1: IF FRONT = -1 OR FRONT > REAR, then

Write UNDERFLOW

Goto Step 2

ELSE

SET VAL = QUEUE[FRONT]

SET FRONT = FRONT + 1

[END OF IF]

Step 2: Exit

LINKED LIST

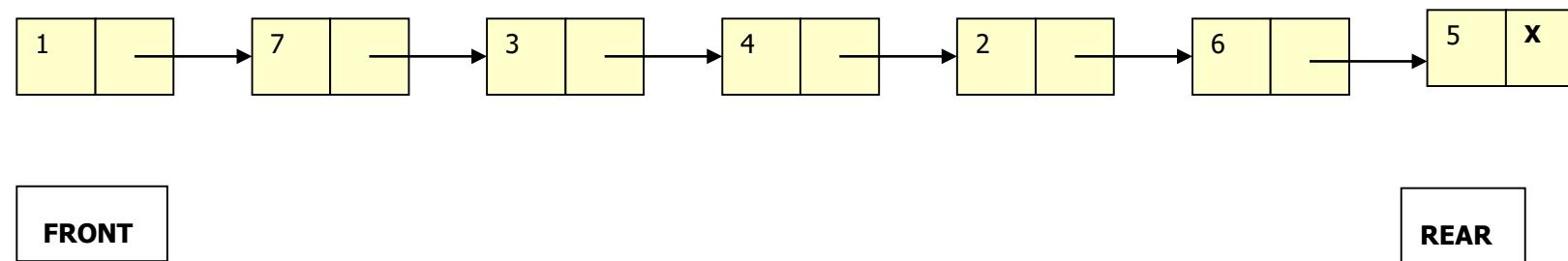
Representation

of

Queues

Linked Representation of Queues

- In a linked queue, every element has two parts: one that stores data and the other that stores the address of the next element.
- The START pointer of the linked list is used as FRONT.
- We will also use another pointer called REAR which will store the address of the last element in the queue.
- All insertions will be done at the rear end and all the deletions will be done at the front end.
- If FRONT = REAR = NULL, then it indicates that the queue is empty.



Inserting an Element in a Linked Queue

Step 1: Allocate memory for the new node and name it as PTR

Step 2: SET PTR->DATA = VAL

Step 3: IF FRONT = NULL, then

SET FRONT = REAR = PTR

SET FRONT->NEXT = REAR->NEXT = NULL

ELSE

SET REAR->NEXT = PTR

SET REAR = PTR

SET REAR->NEXT = NULL

[END OF IF]

Step 4: END

Deleting an Element from a Linked Queue

Step 1: IF FRONT = NULL, then

 Write “Underflow”

 Go to Step 5

 [END OF IF]

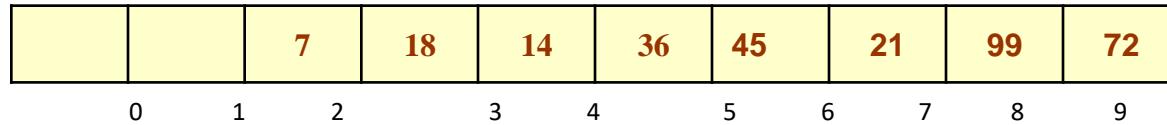
Step 2: SET PTR = FRONT

Step 3: FRONT = FRONT->NEXT

Step 4: FREE PTR

Step 5: END

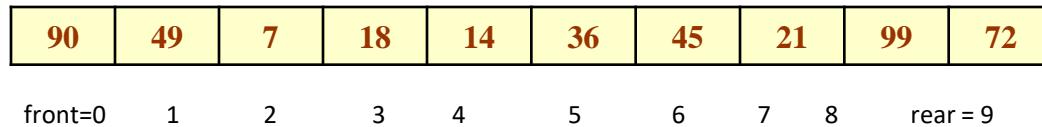
Circular Queues



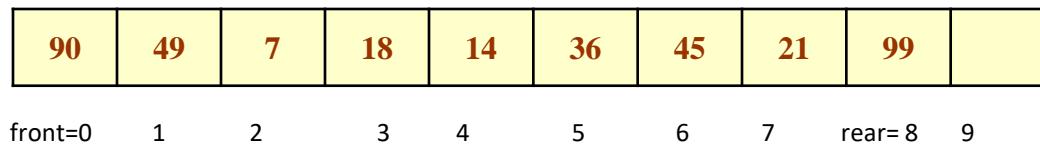
- We will explain the concept of circular queues using an example.
- In this queue, front = 2 and rear = 9.
- Now, if you want to insert a new element, it cannot be done because the space is available only at the left of the queue.
- If $\text{rear} = \text{MAX} - 1$, then OVERFLOW condition exists.
- This is the major drawback of a linear queue. Even if space is available, no insertions can be done once rear is equal to $\text{MAX} - 1$.
- This leads to wastage of space. In order to overcome this problem, we use circular queues.
- In a circular queue, the first index comes right after the last index.
- A circular queue is full, only when $\text{front}=0$ and $\text{rear} = \text{Max} - 1$.

Inserting an Element in a Circular Queue

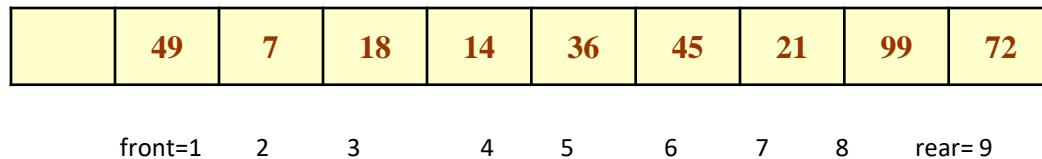
- For insertion we check for three conditions which are as follows:
 - If $\text{front}=0$ and $\text{rear}= \text{MAX} - 1$, then the circular queue is full.



- If $\text{rear} \neq \text{MAX} - 1$, then the rear will be incremented and value will be inserted



- If $\text{front} \neq 0$ and $\text{rear}=\text{MAX} - 1$, then it means that the queue is not full. So, set $\text{rear} = 0$ and insert the new element.



Algorithm to Insert an Element in a Circular Queue

Step 1: IF FRONT = 0 and Rear = MAX - 1, then

 Write "OVERFLOW"

 Goto Step 4

 [END OF IF]

Step 2: IF FRONT = -1 and REAR = -1, then;

 SET FRONT = REAR = 0

 ELSE IF REAR = MAX - 1 and FRONT != 0

 SET REAR = 0

 ELSE

 SET REAR = REAR + 1

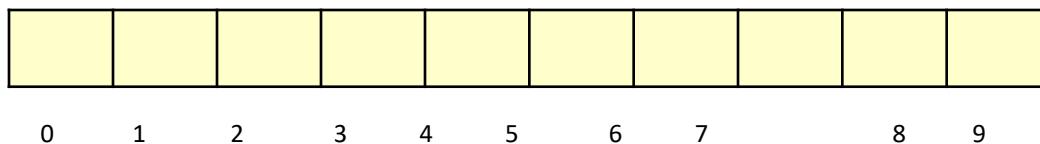
 [END OF IF]

Step 3: SET QUEUE [REAR] = VAL

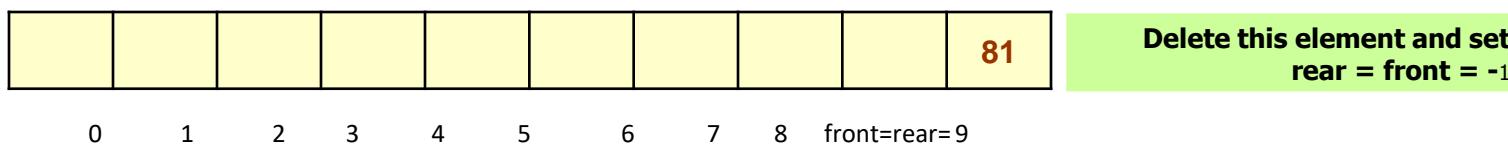
Step 4: Exit

Deleting an Element from a Circular Queue

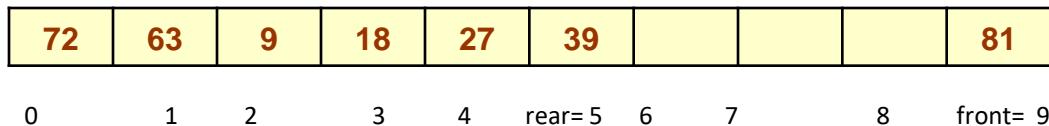
- To delete an element again we will check for three conditions:
- If front = -1, then it means there are no elements in the queue. So an underflow condition will be reported.



- If the queue is not empty and after returning the value on front, if front = rear, then it means now the queue has become empty and so front and rear are set to -1.



- If the queue is not empty and after returning the value on front, if front = MAX - 1, then front is set to 0.



Algorithm to Delete an Element from a Circular Queue

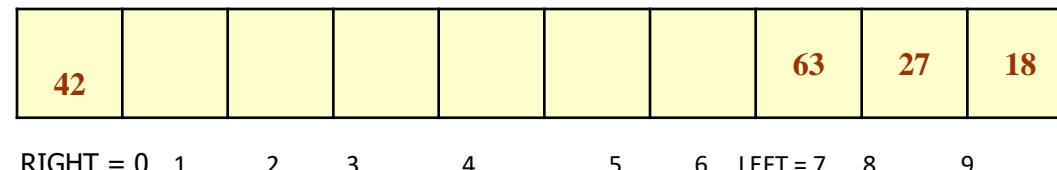
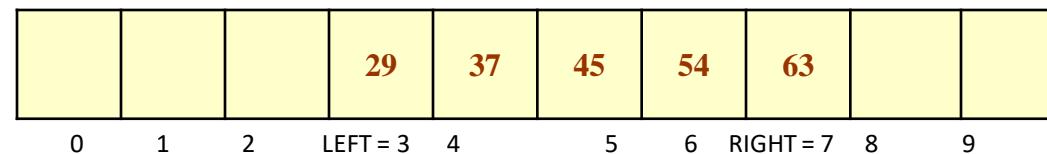
```
Step 1: IF FRONT = -1, then
            Write "Underflow"
            Goto Step 4
            [END OF IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
        ELSE
            IF FRONT = MAX -1
                SET FRONT = 0
            ELSE
                SET FRONT = FRONT + 1
            [END OF IF]
        [END OF IF]
Step 4: EXIT
```

Deques

- A deque is a list in which elements can be inserted or deleted at either end.
- It is also known as a head-tail linked list because elements can be added to or removed from the front (head) or back (tail).
- A deque can be implemented either using a circular array or a circular doubly linked list.
- In a deque, two pointers are maintained, LEFT and RIGHT which point to either end of the deque.
- The elements in a deque stretch from LEFT end to the RIGHT and since it is circular, Dequeue[N-1] is followed by Dequeue[0].

Deques

- There are two variants of a double-ended queue:
 - *Input restricted deque*: In this dequeue insertions can be done only at one of the ends while deletions can be done from both the ends.
 - *Output restricted deque*: In this dequeue deletions can be done only at one of the ends while insertions can be done on both the ends.



Priority Queues

- A priority queue is a queue in which each element is assigned a priority.
- The priority of elements is used to determine the order in which these elements will be processed.
- The general rule of processing elements of a priority queue can be given as:
 - An element with higher priority is processed before an element with lower priority
 - Two elements with same priority are processed on a first come first served (FCFS) basis
- Priority queues are widely used in operating systems to execute the highest priority process first.
- In computer's memory priority queues can be represented using arrays or linked lists.

Array Representation of Priority Queues

- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained.
- Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.
- We can use a two-dimensional array for this purpose where each queue will be allocated same amount of space.
- Given the front and rear values of each queue, a two dimensional matrix can be formed.

FRONT	REAR
3	3
1	3
4	5
4	1

1	2	3	4	5
1		A		
2	B	C	D	
3			E	F
4	I		G	H

Priority queue matrix

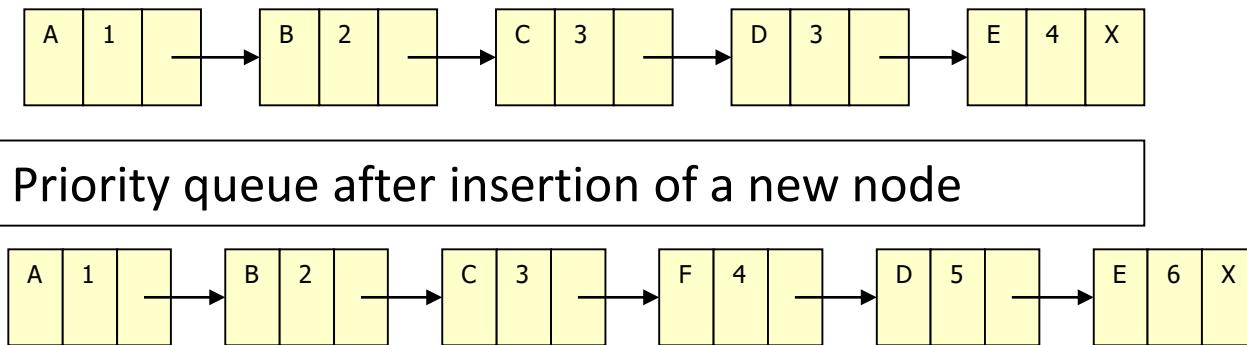
FRONT	REAR
3	3
1	3
4	1
4	1

1	2	3	4	5
1		A		
2	B	C	D	
3	R		E	F
4	I		G	H

Priority queue matrix after insertion of a new element

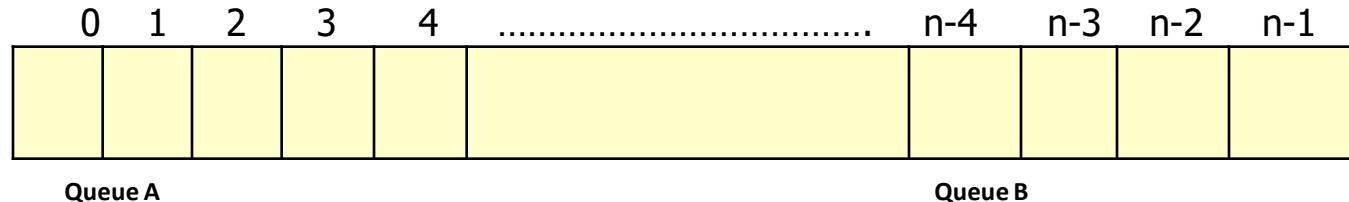
Linked Representation of Priority Queues

- When a priority queue is implemented using a linked list, then every node of the list contains three parts: (1) the information or data part, (ii) the priority number of the element, (iii) and address of the next element.
- If we are using a sorted linked list, then element having higher priority will precede the element with lower priority.



Multiple Queues

- When implementing a queue using an array, the size of the array must be known in advance.
- If the queue is allocated less space, then frequent OVERFLOW conditions will be encountered.
- To deal with this problem, the code will have to be modified to reallocate more space for the array, but this results in sheer wastage of memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.
- A better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array.
- One important point to note is that while queue A will grow from left to right, the queue B on the same time will grow from right to left.



Applications of Queues

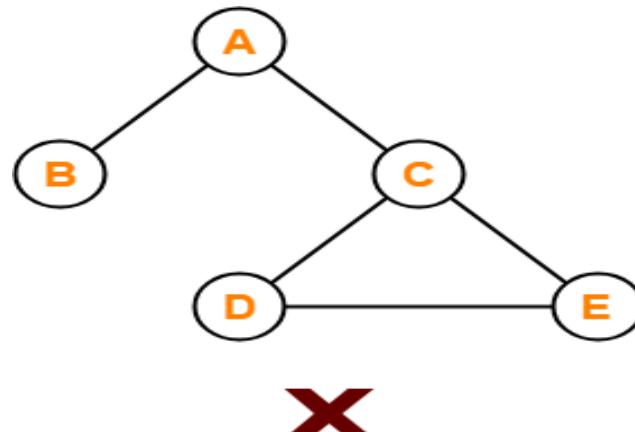
- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used to transfer data asynchronously e.g., pipes, file IO, sockets.
- Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- Queues are used in OS for handling interrupts. When programming a real-time system that can be interrupted, for ex, by a mouse click, it is necessary to process the interrupts immediately before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure

UNIT 3

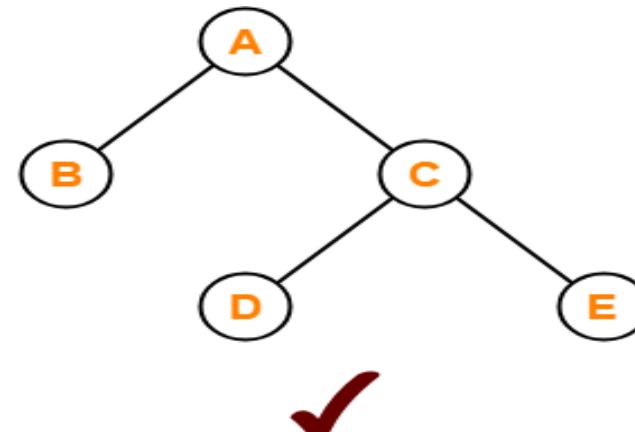
TREES

Definition

- Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive.
- A tree is a connected graph without any circuits.
- If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.



This graph is not a Tree

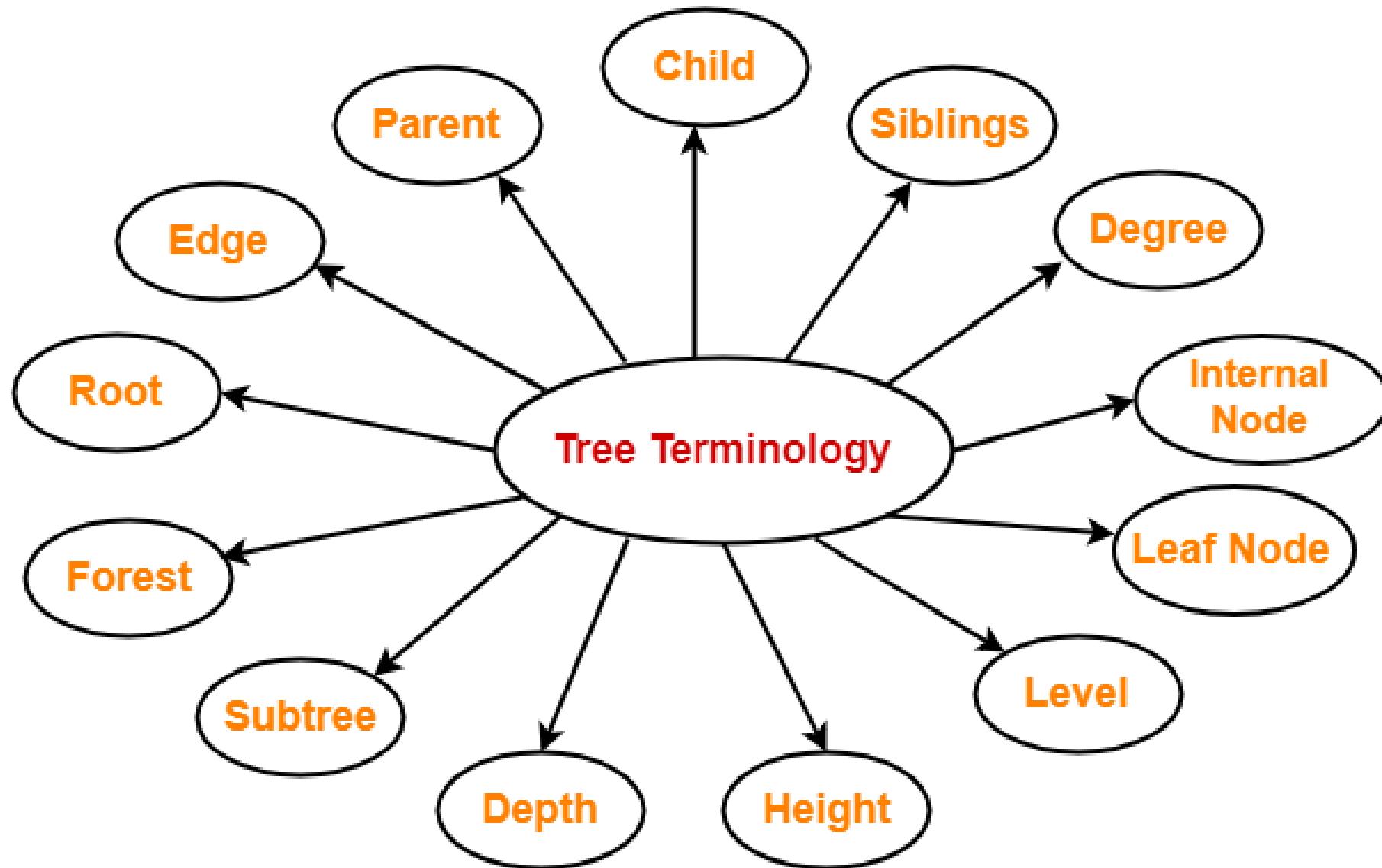


This graph is a Tree

Properties of Trees

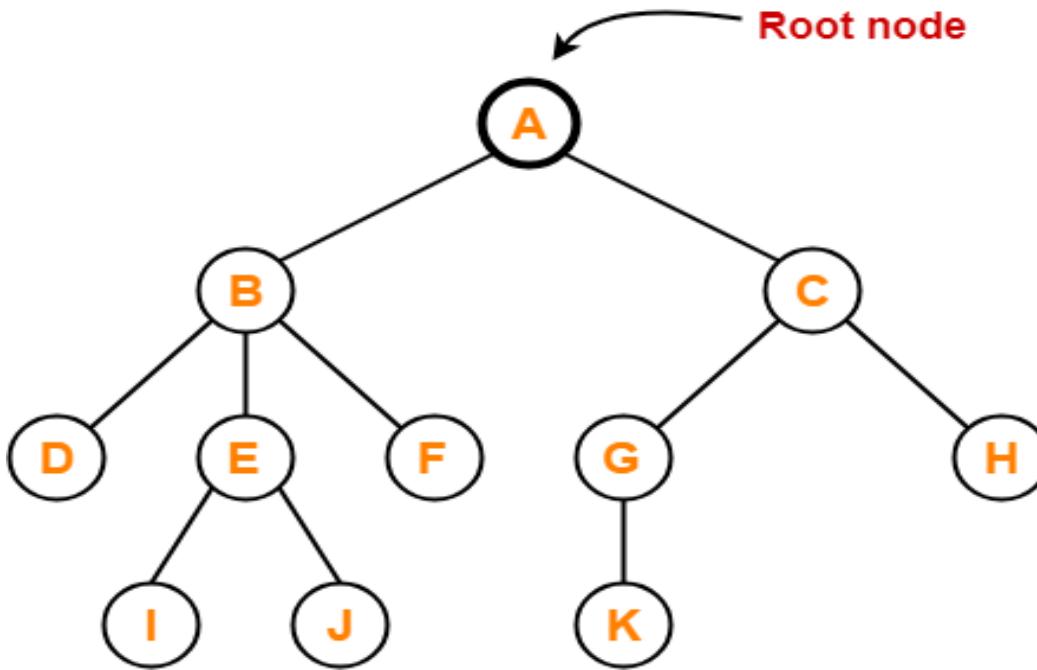
- ✓ There is one and only one path between every pair of vertices in a tree.
- ✓ A tree with n vertices has exactly $(n-1)$ edges.
- ✓ A graph is a tree if and only if it is minimally connected.
- ✓ Any connected graph with n vertices and $(n-1)$ edges is a tree.

Basic Tree Terminology



Root

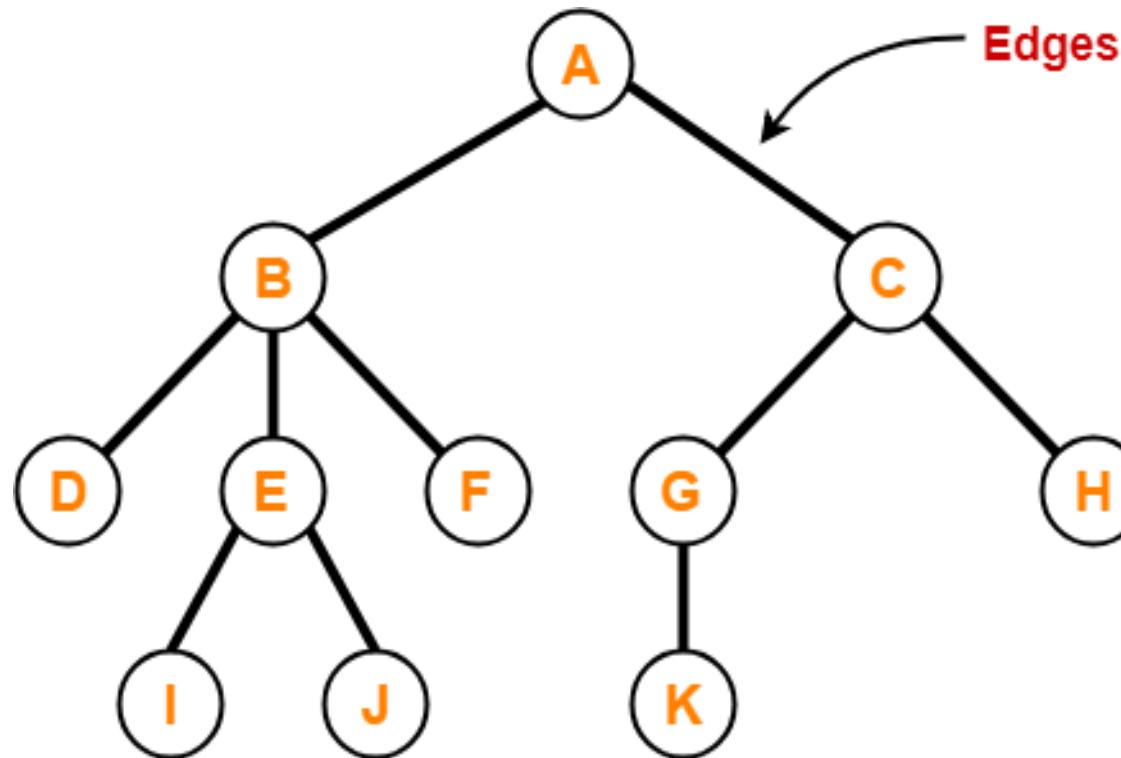
- ✓ The first node from where the tree originates is called as a root node.
- ✓ In any tree, there must be only one root node.
- ✓ We can never have multiple root nodes in a tree data structure.



Here, node **A** is the only root node

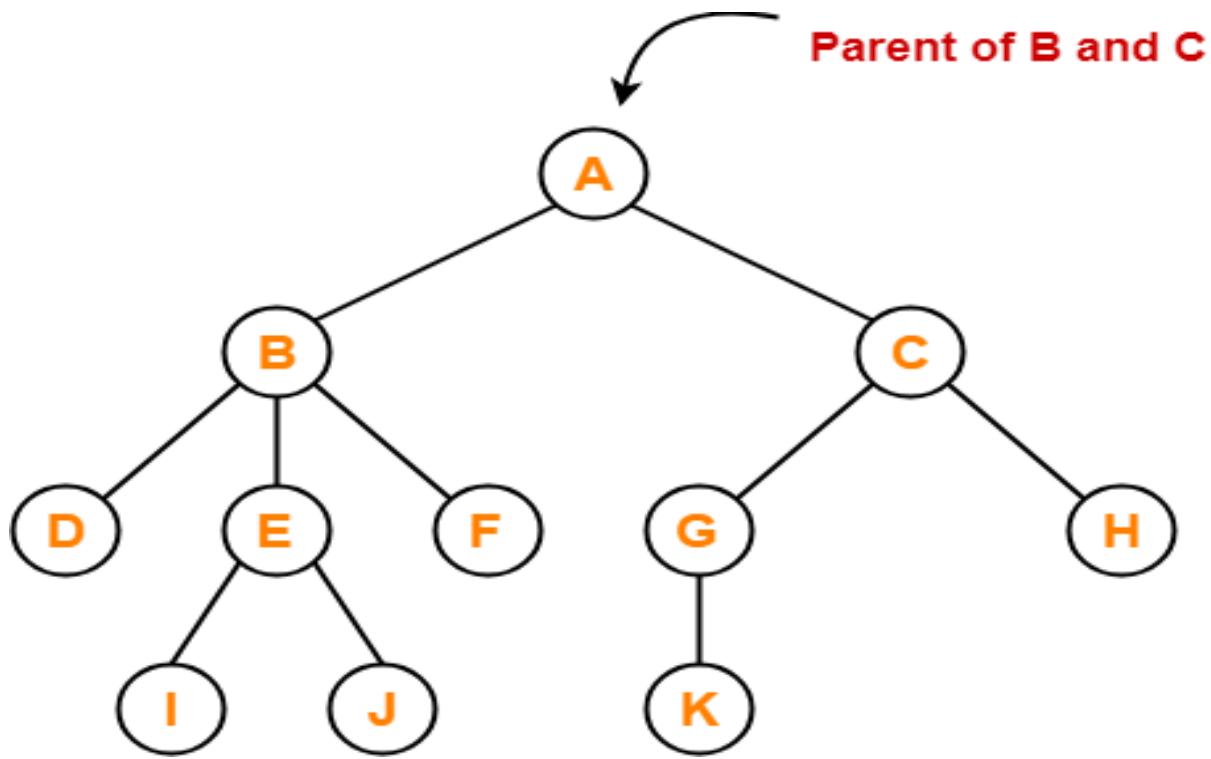
Edge

- ✓ The connecting link between any two nodes is called as an edge.
- ✓ In a tree with n number of nodes, there is exactly $(n-1)$ number of edges.



Parent Nodes

- ✓ The node which has a branch from it to any other node is called as a parent node.
- ✓ In other words, the node which has one or more children is called as a parent node.
- ✓ In a tree, a parent node can have any number of child nodes.

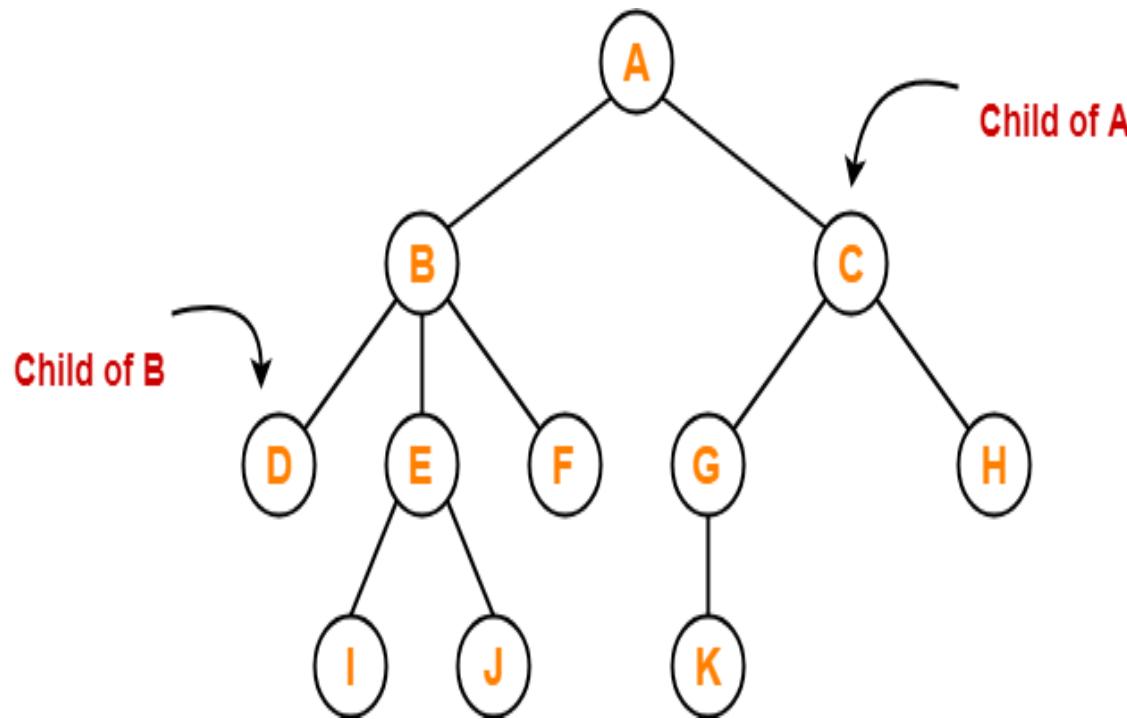


Here,

- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

Child Nodes

- ✓ The node which is a descendant of some node is called as a child node.
- ✓ All the nodes except root node are child nodes.

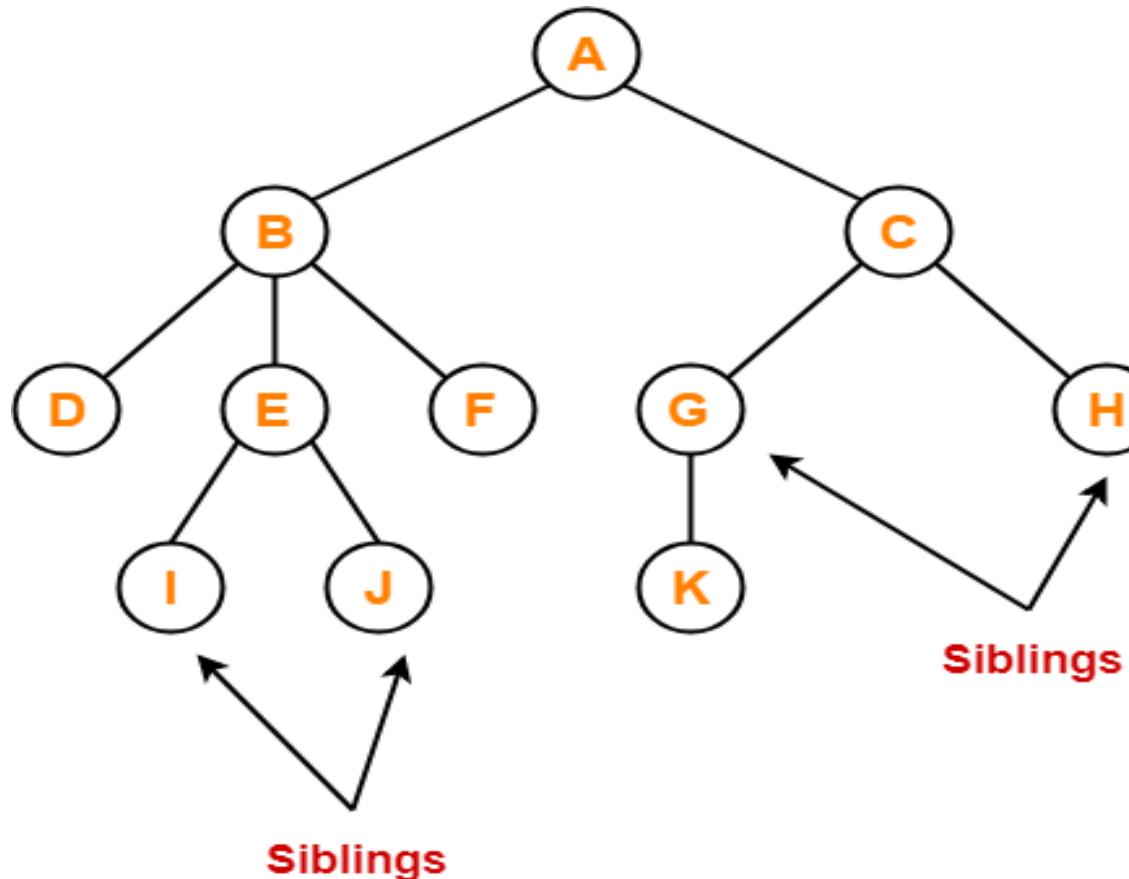


Here,

- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

Siblings

- ✓ Nodes which belong to the same parent are called as siblings.
- ✓ In other words, nodes with the same parent are sibling nodes.

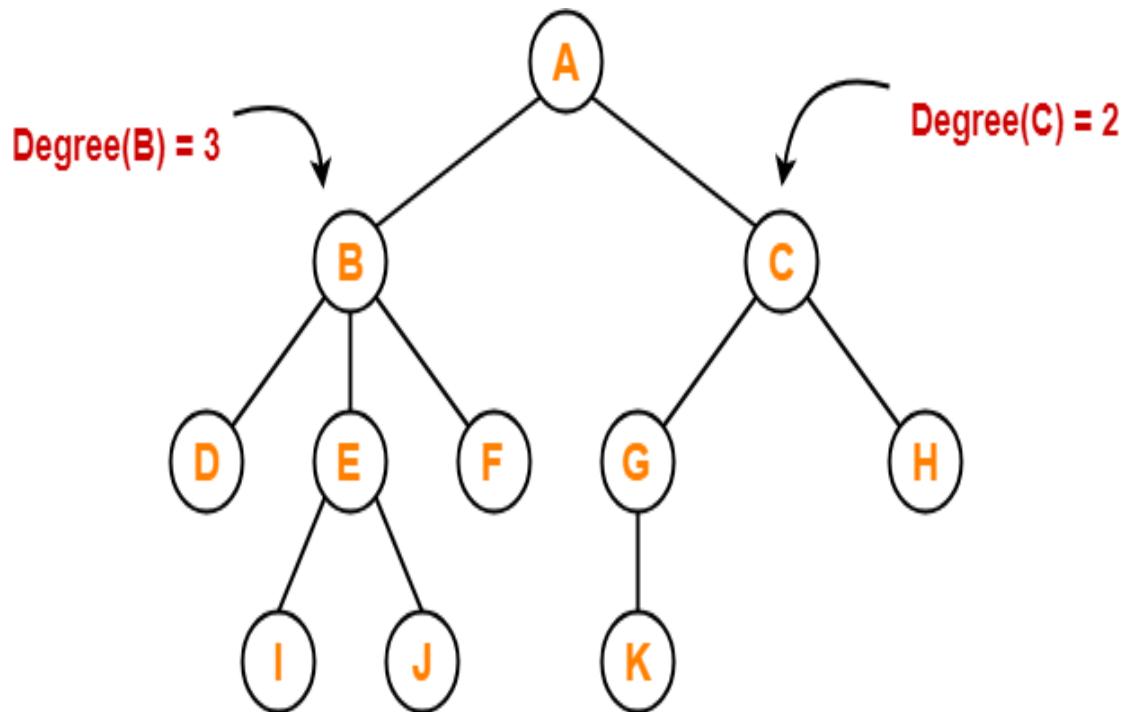


Here,

- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

Degree of a Node

- ✓ Degree of a node is the total number of children of that node.
- ✓ Degree of a tree is the highest degree of a node among all the nodes in the tree.

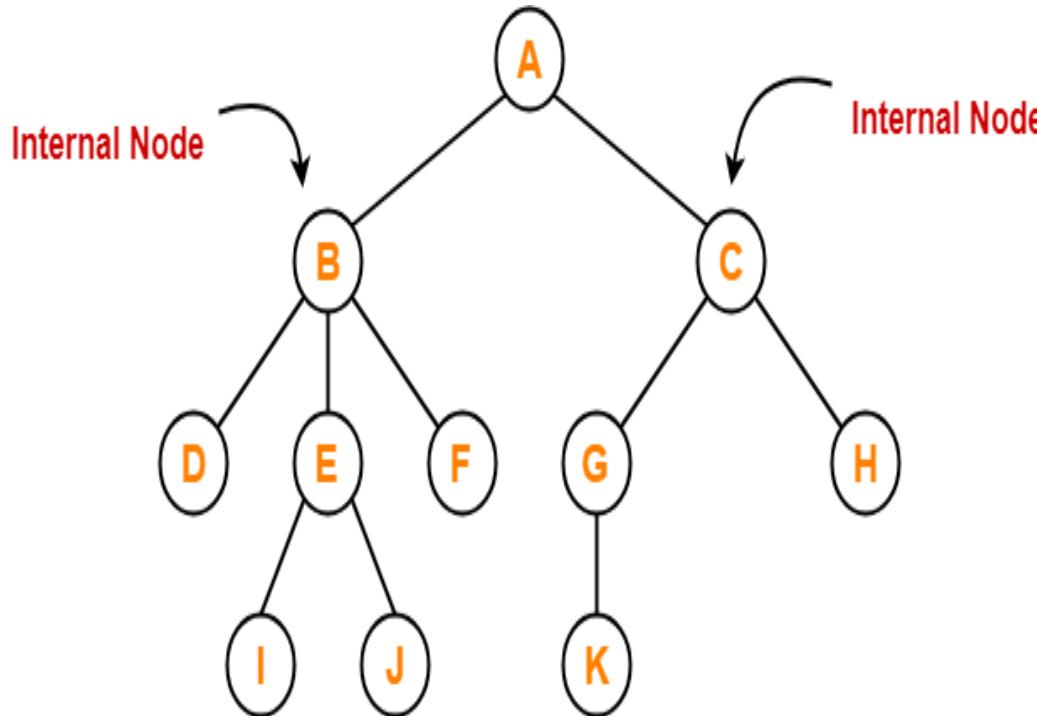


Here,

- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

Internal Node

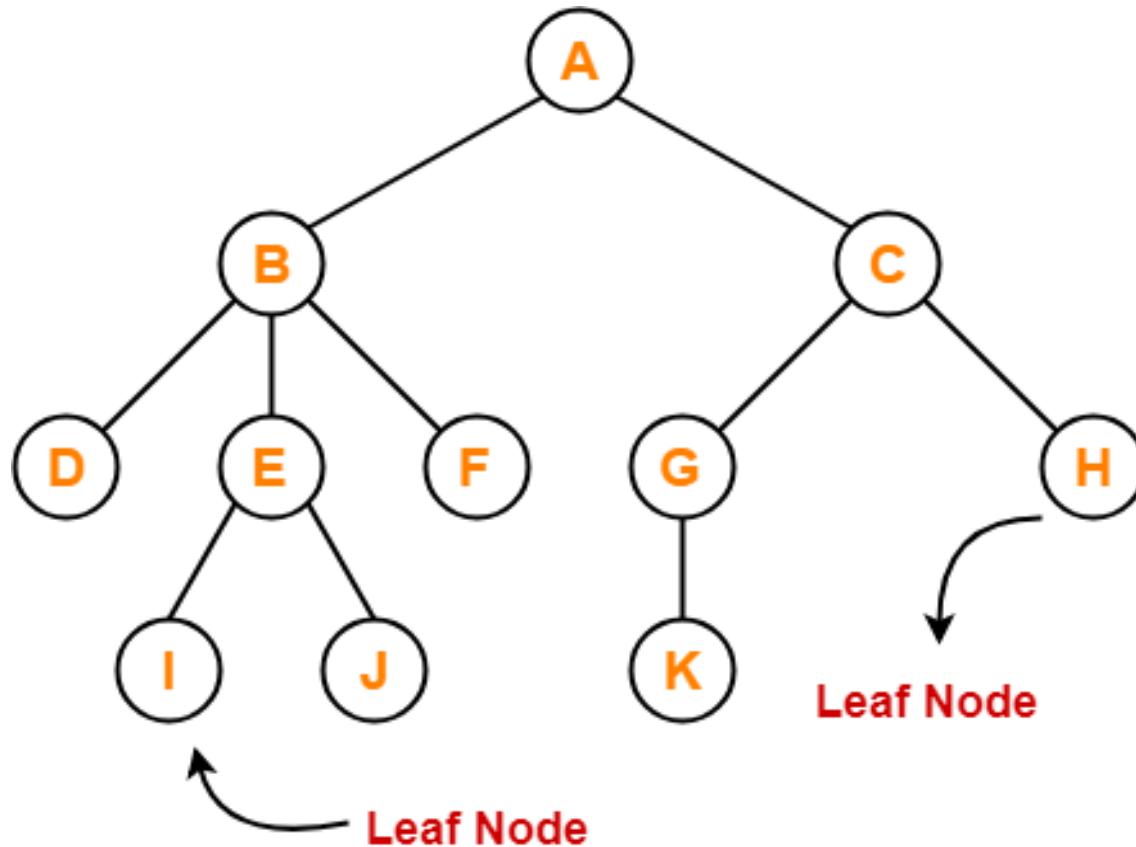
- ✓ The node which has at least one child is called as an internal node.
- ✓ Internal nodes are also called as non-terminal nodes.
- ✓ Every non-leaf node is an internal node.



Here, nodes **A, B, C, E** and **G** are internal nodes.

Leaf Node

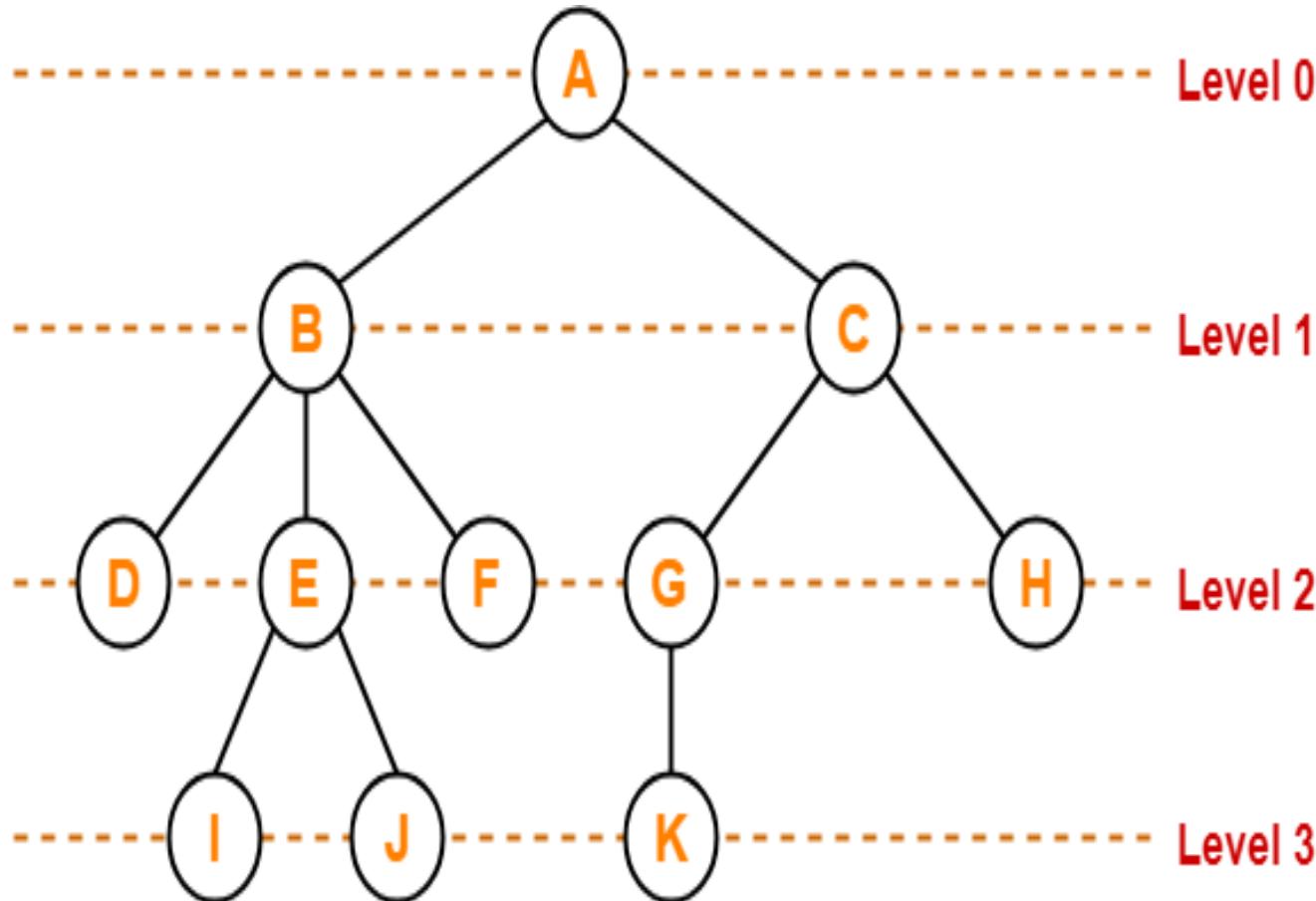
- ✓ The node which does not have any child is called as a leaf node.
- ✓ Leaf nodes are also called as external nodes or terminal nodes.



Here, nodes **D, I, J, F, K and H** are leaf nodes.

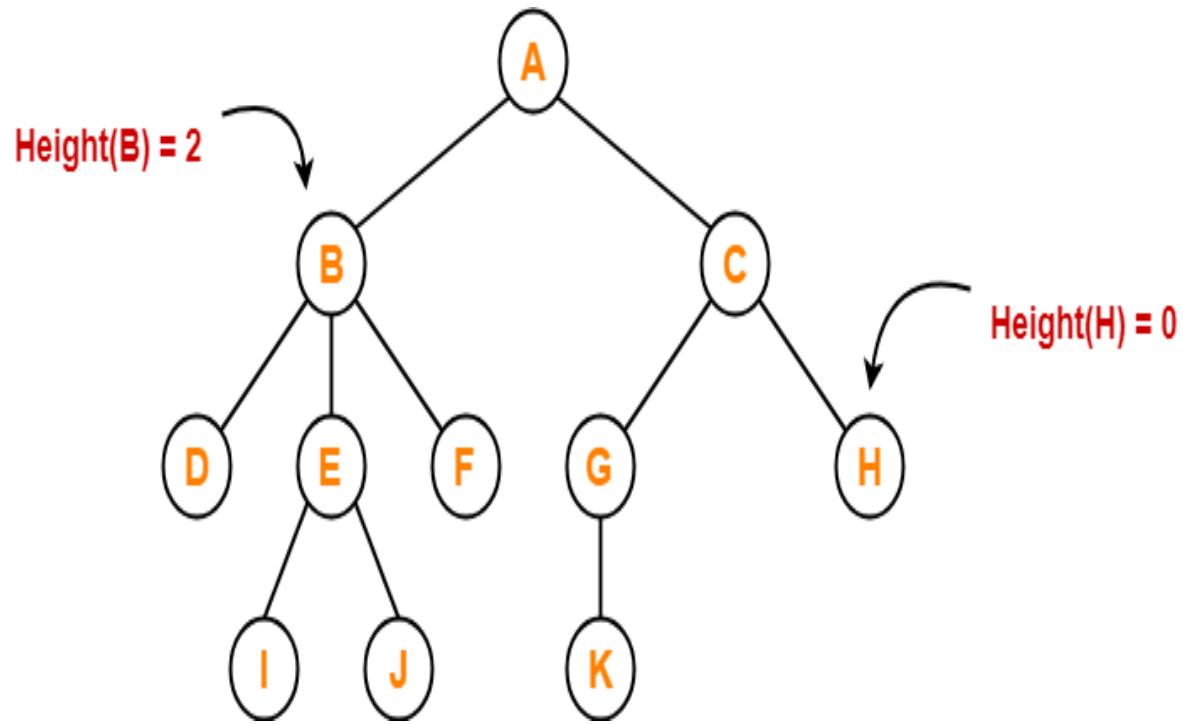
Level

- ✓ In a tree, each step from top to bottom is called as level of a tree.
- ✓ The level count starts with 0 and increments by 1 at each level or step.



Height

- ✓ Total number of edges that lies on the longest path from any leaf node to a particular node is called as height of that node.
- ✓ Height of a tree is the height of root node.
- ✓ Height of all leaf nodes = 0

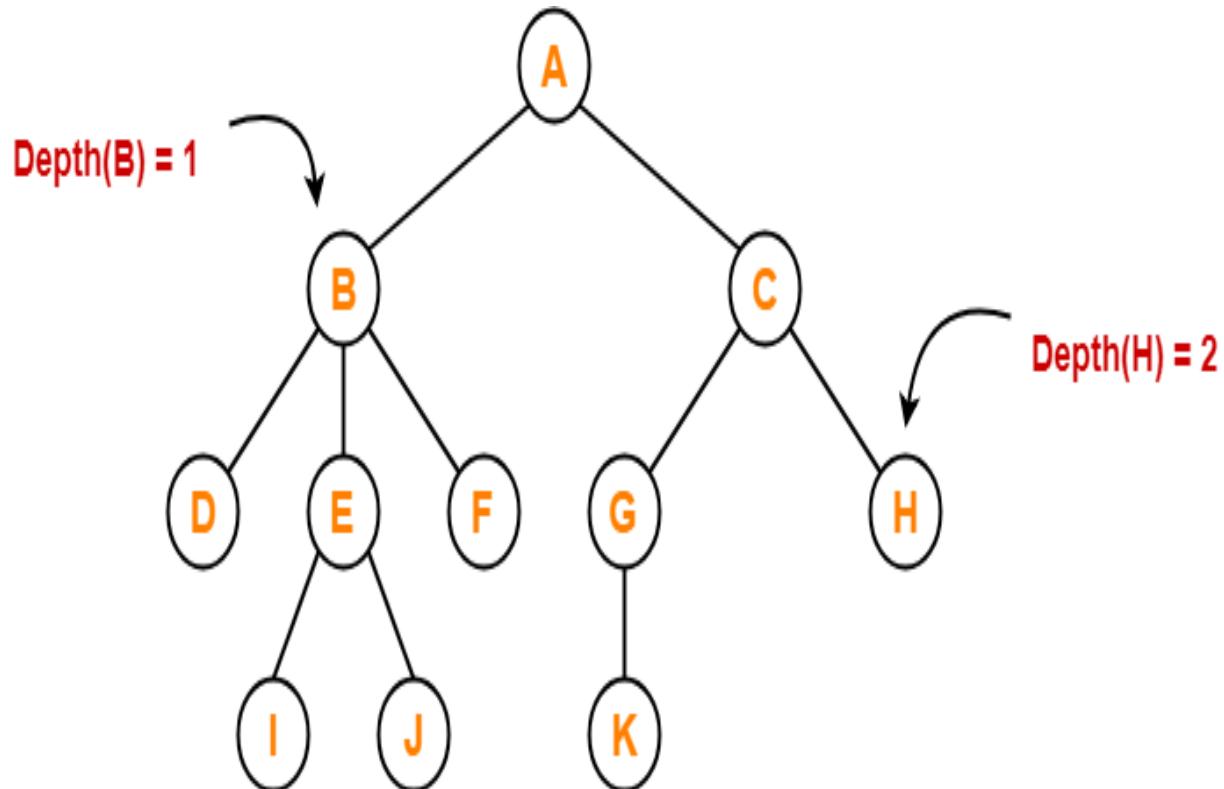


Here

- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

Depth

- ✓ Total number of edges from root node to a particular node is called as depth of that node.
- ✓ Depth of a tree is the total number of edges from root node to a leaf node in the longest path.
- ✓ Depth of the root node = 0
- ✓ The terms “level” and “depth” are used interchangeably.

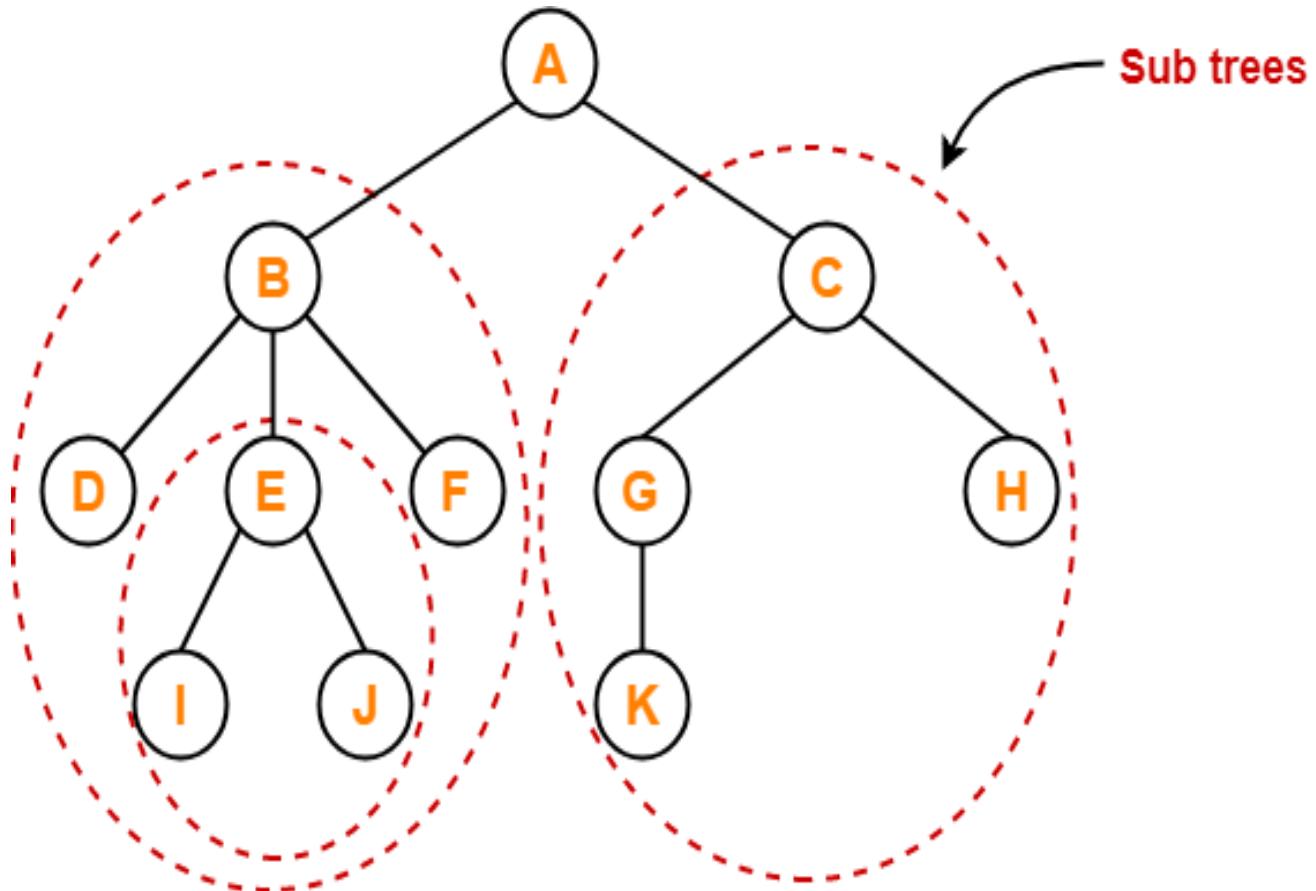


Here

- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3

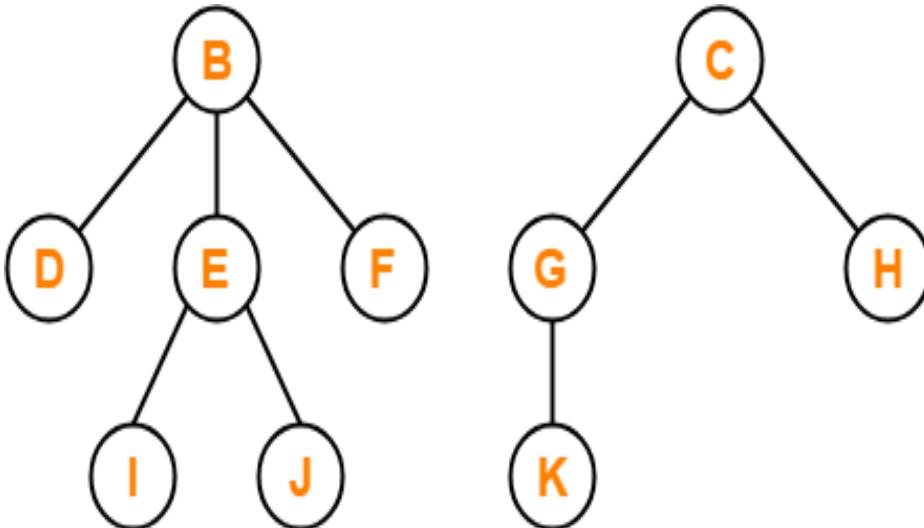
Sub-tree

- ✓ In a tree, each child from a node forms a sub-tree recursively.
- ✓ Every child node forms a sub-tree on its parent node.



Forest

- ✓ A forest is a set of disjoint trees.



Forest

Advantages of Tree

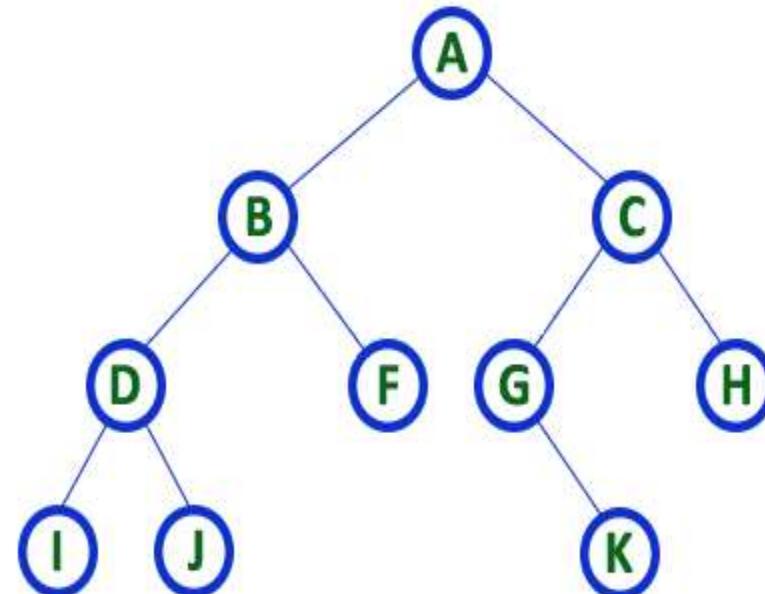
- ✓ Tree reflects structural relationships in the data.
- ✓ It is used to represent hierarchies.
- ✓ It provides an efficient insertion and searching operations.
- ✓ Trees are flexible. It allows to move sub trees around with minimum effort.

Tree Representation

A tree data structure is represented using two methods.

✓ Array Representation

✓ Linked List Representation



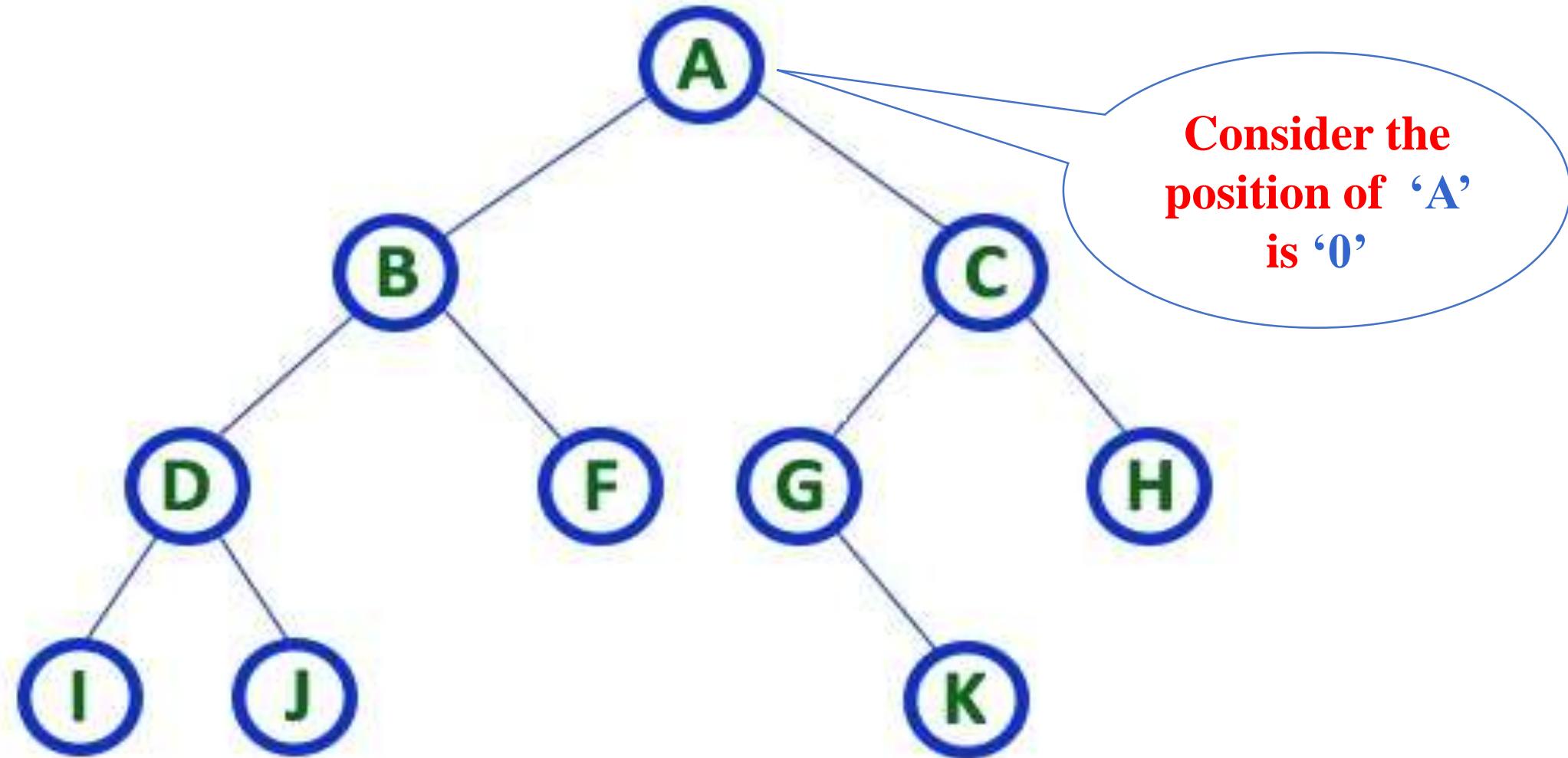
Array Representation

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

- Consider the root node at index ‘0’
- LEFT child is placed at $2i+1$
- RIGHT child is placed at $2i+2$

Where ‘i’ is the position of the parent

Example

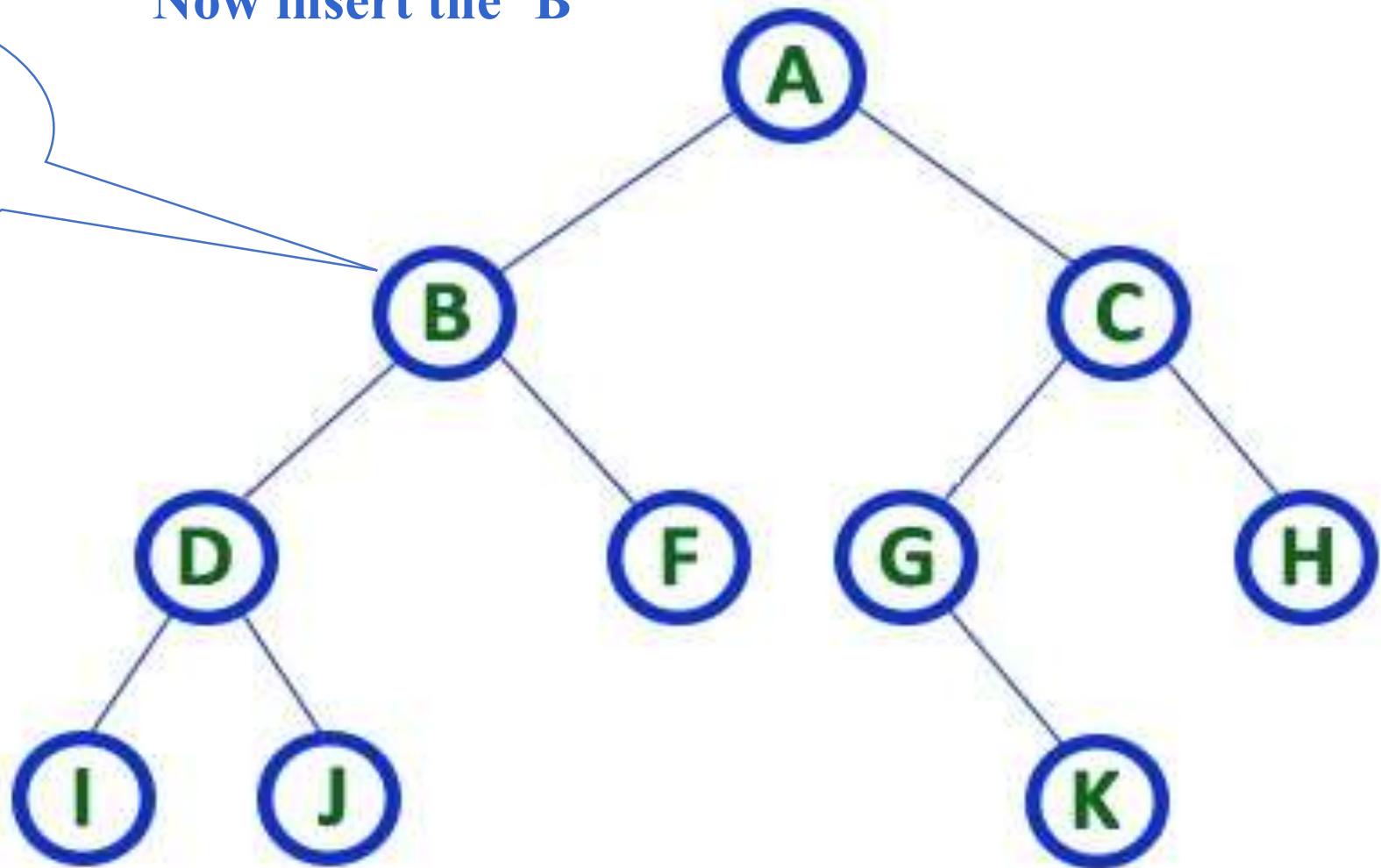


**Array
Index**

A															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now inserting
the 'B'

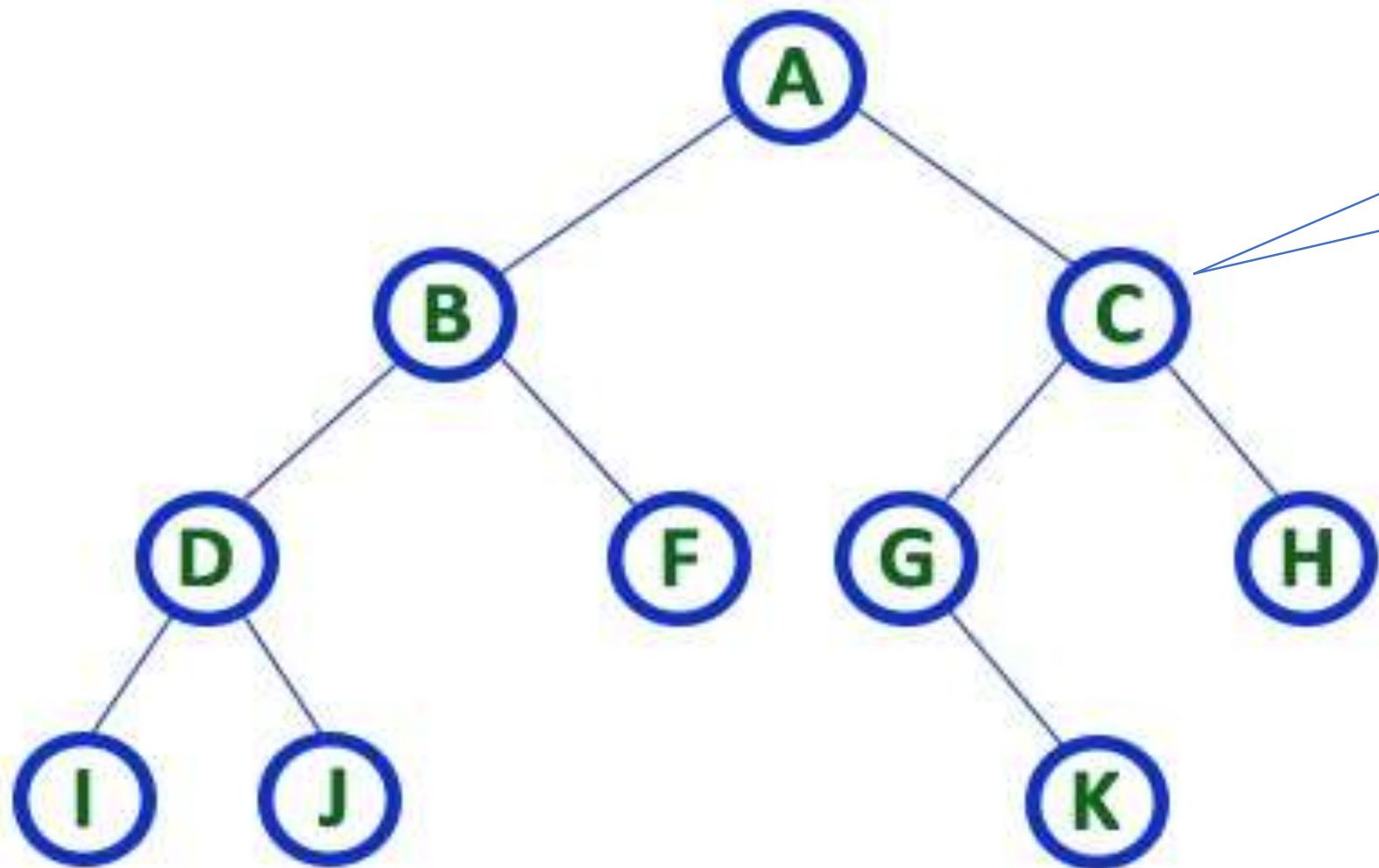
Now insert the 'B'



Array
Index

A	B														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the 'C'



Now inserting
the 'C'

where Pos of 'A' is '0'
'C' is a Right child of
'A' then formula is

$$\begin{aligned} & 2i+2 \\ & = 2(0)+2 \\ & = 2 \end{aligned}$$

Array
Index

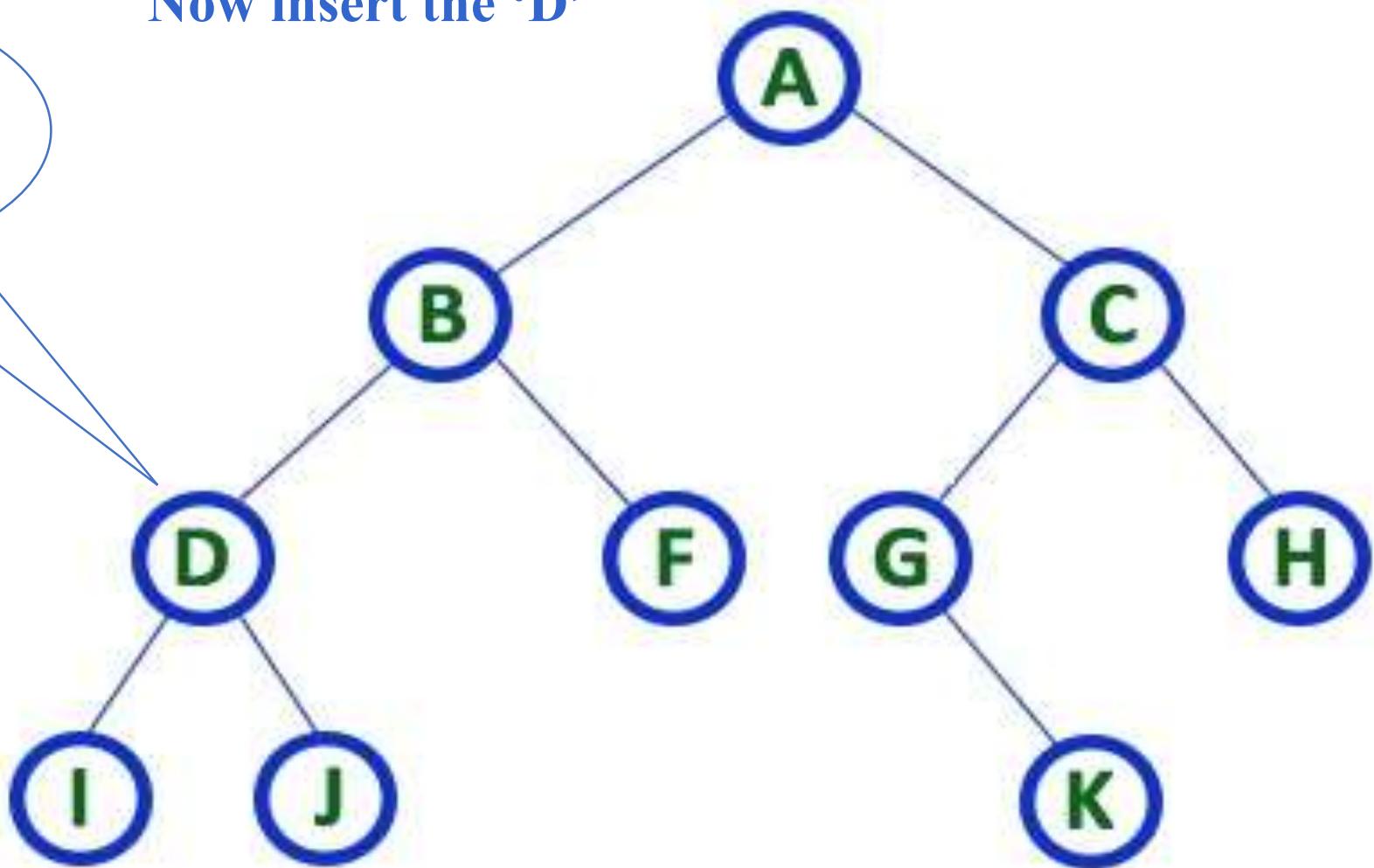
A	B	C													
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now inserting
the 'D'

where Pos of 'B' is '1'
'D' is a left child of 'B'
then formula is $2i+1$

$$2(1)+1
= 3$$

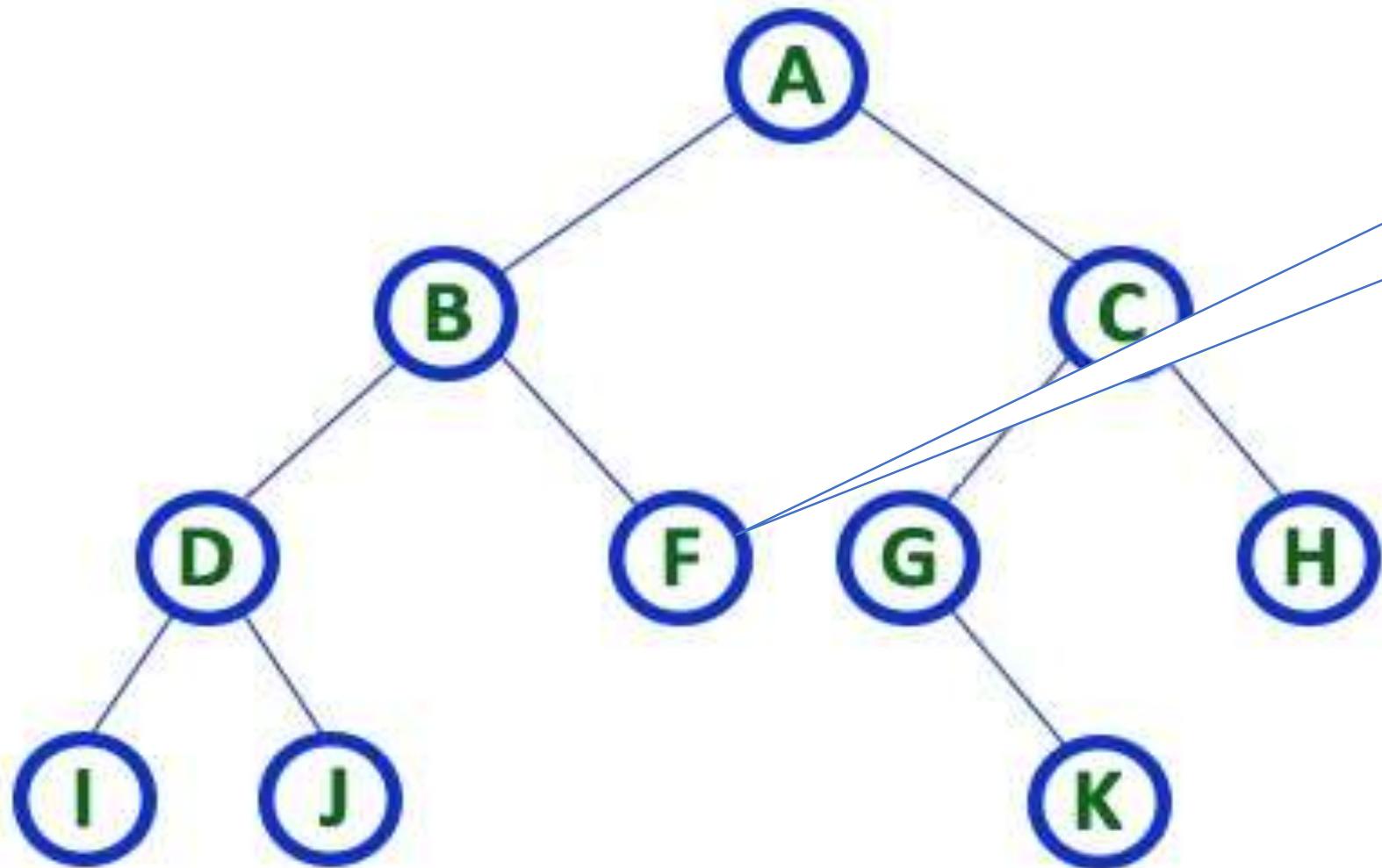
Now insert the 'D'



Array
Index

A	B	C	D												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the 'F'



Now inserting
the 'F'

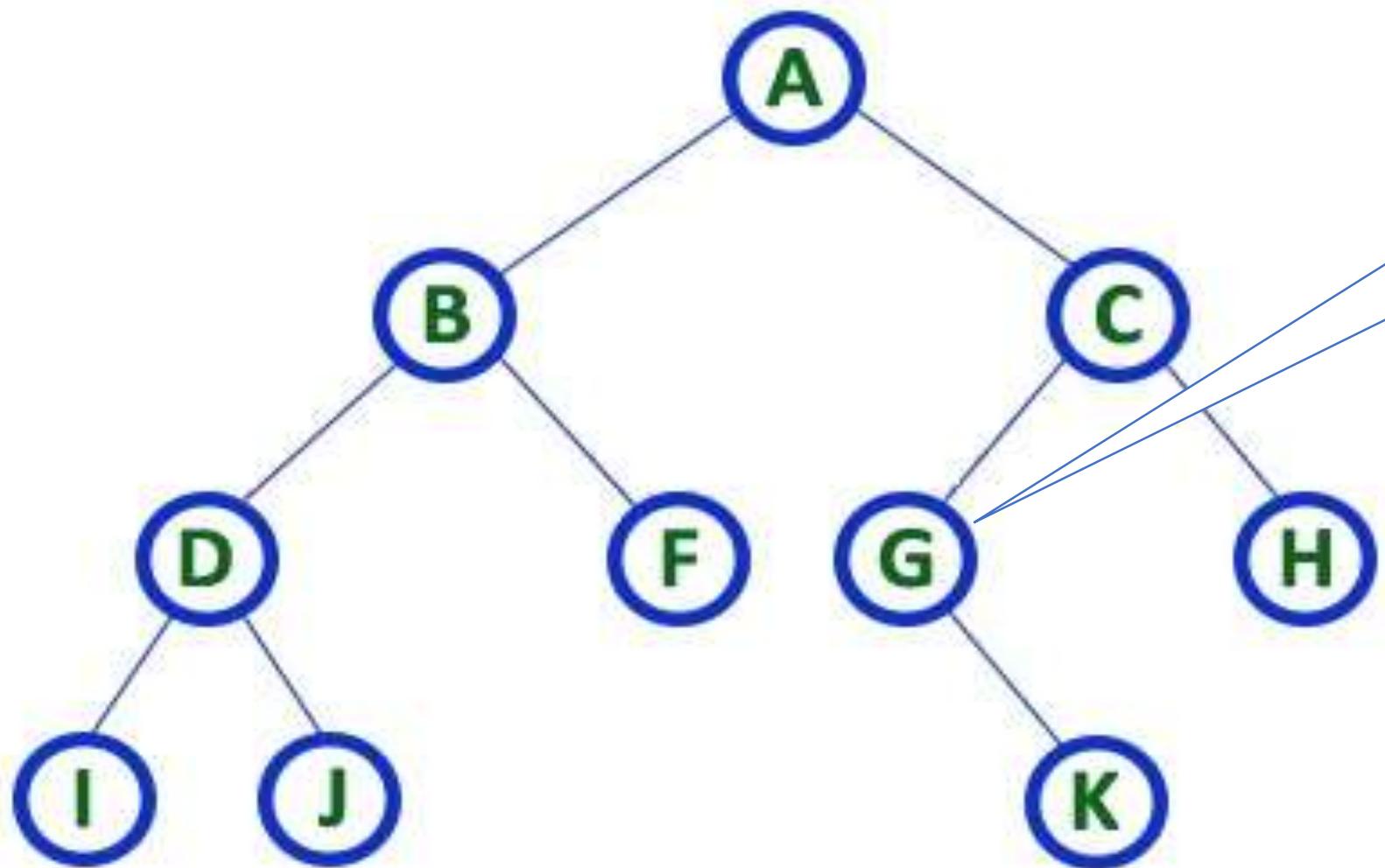
where Pos of 'B' is '1'
'F' is a Right child of
'B' then formula is

$$\begin{aligned}2i+2 \\= 2(1)+2 \\= 4\end{aligned}$$

Array
Index

A	B	C	D	F											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the 'G'



Now inserting
the 'G'

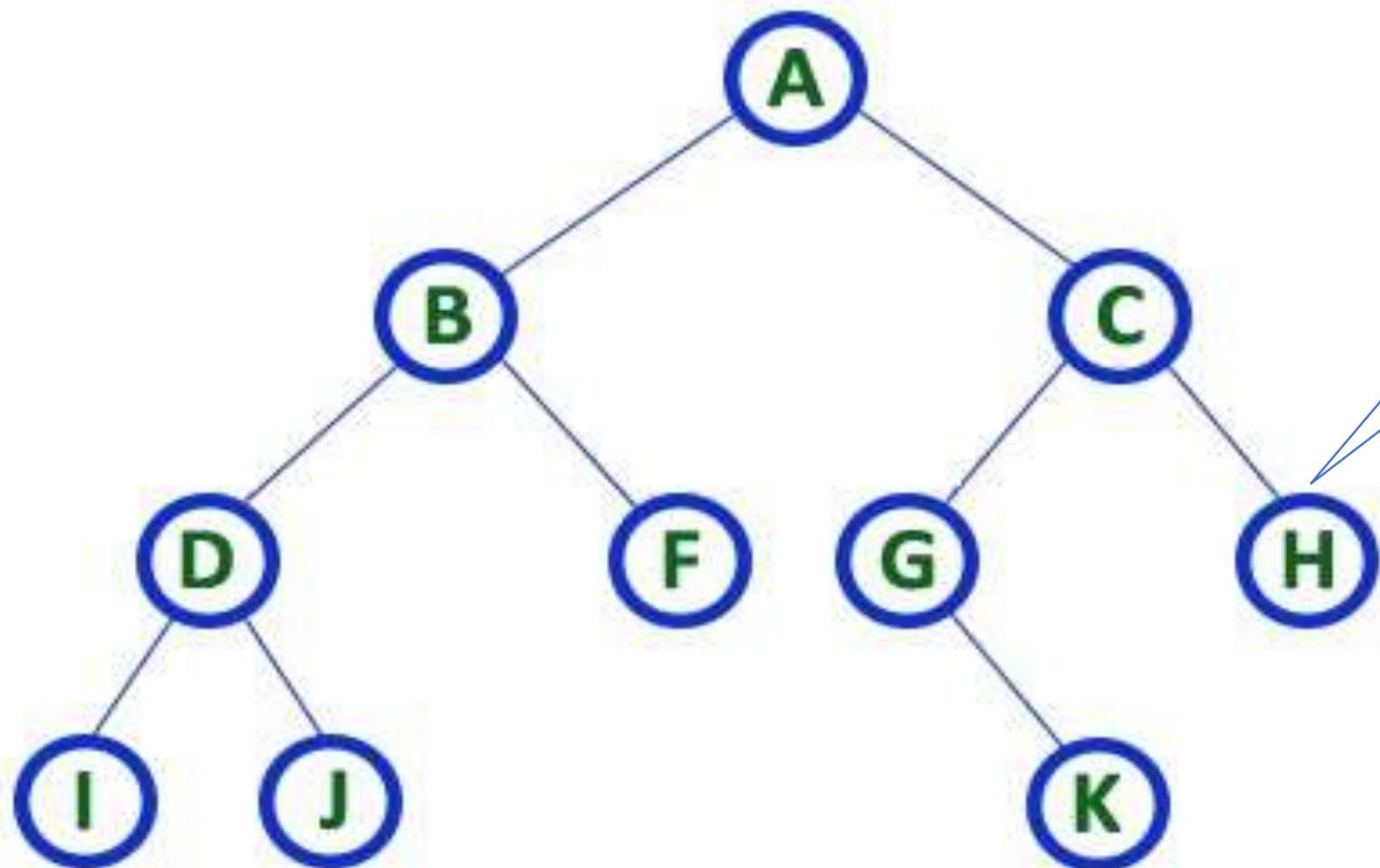
where Pos of 'C' is '2'
'G' is a left child of 'C'
then formula is $2i+1$

$$\begin{aligned} &= 2(2)+1 \\ &= 5 \end{aligned}$$

Array
Index

A	B	C	D	F	G										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the 'H'



Now inserting
the 'H'

where Pos of 'C' is '2'
'H' is a Right child of
'C' then formula is

$$2i+2$$

$$= 2(2)+2
= 6$$

Array
Index

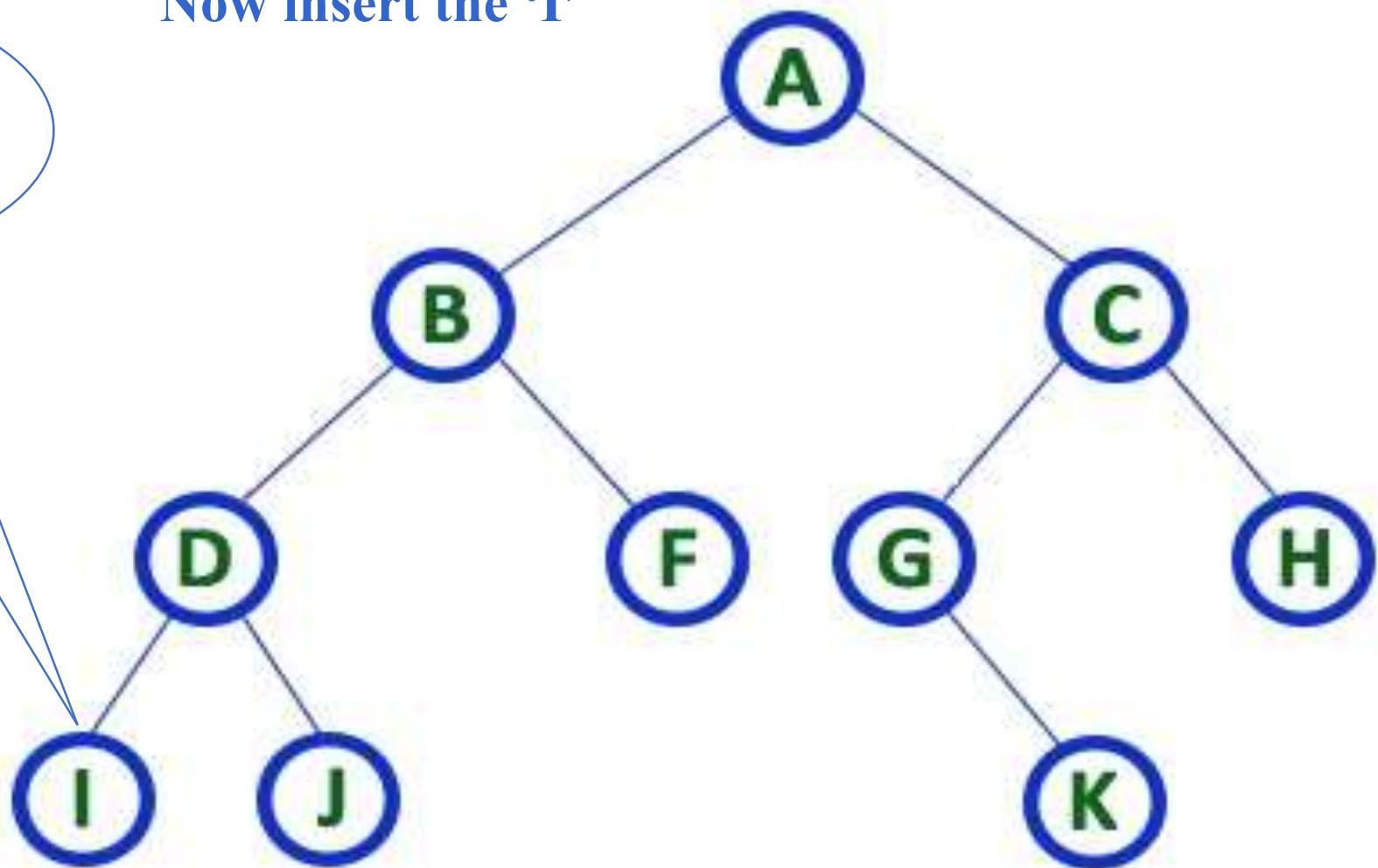
A	B	C	D	F	G	H									
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now inserting
the 'I'

where Pos of 'D' is '3'
 'I' is a left child of 'D'
 then formula is $2i+1$

$$2(3)+1
= 7$$

Now insert the 'I'



**Array
Index**

A	B	C	D	F	G	H	I							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

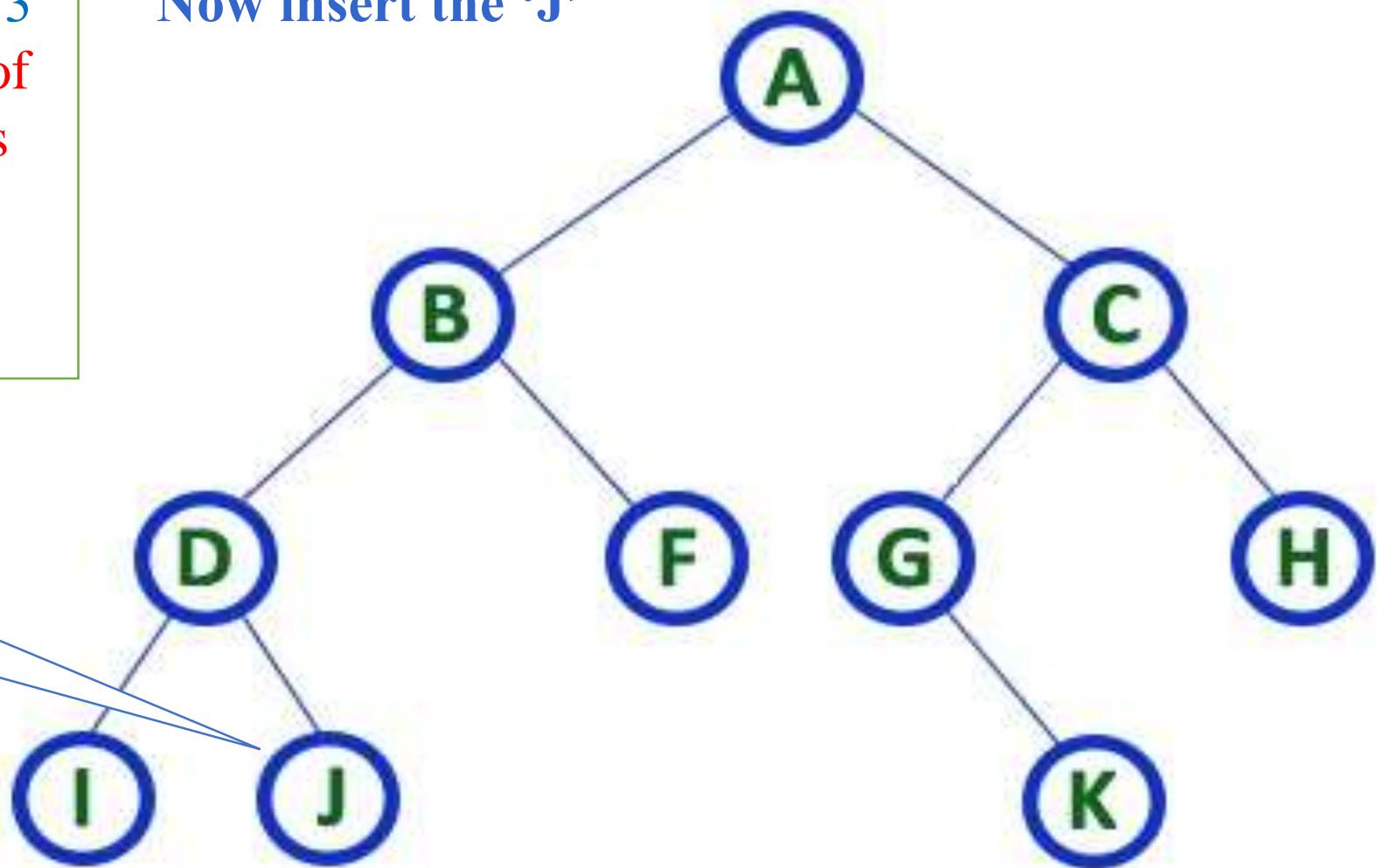
where Pos of 'D' is '3'
'J' is a Right child of
'D' then formula is

$$2i+2$$

$$\begin{aligned}2(3)+2 \\= 8\end{aligned}$$

Now
inserting the
'J'

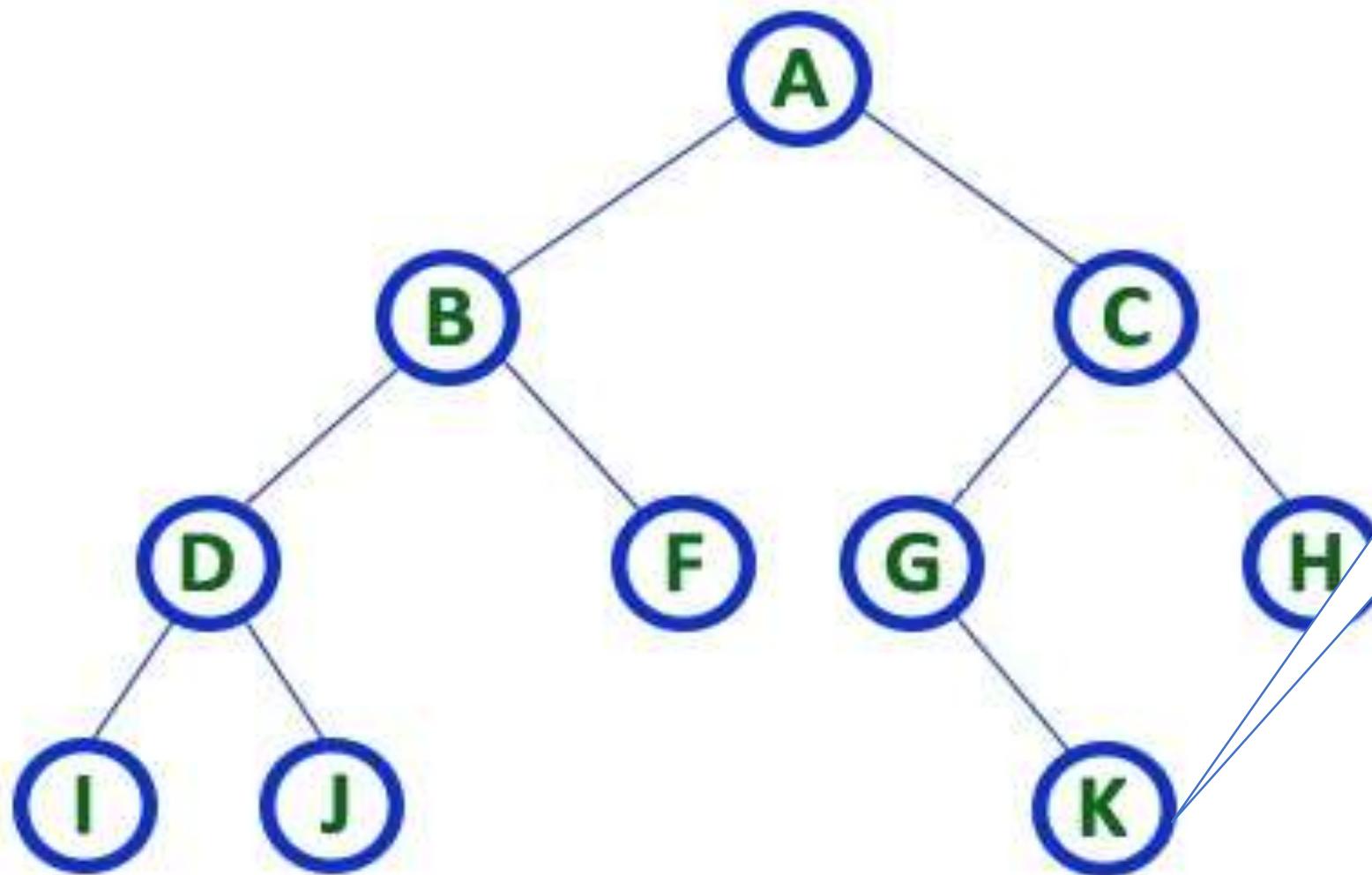
Now insert the 'J'



Array
Index

A	B	C	D	F	G	H	I	J							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the 'K'



Now inserting
the 'K'

where Pos of 'G' is '5'
'K' is a Right child of
'G' then formula is

$$\begin{aligned} & 2i+2 \\ & = 2(5)+2 \\ & = 12 \end{aligned}$$

Array
Index

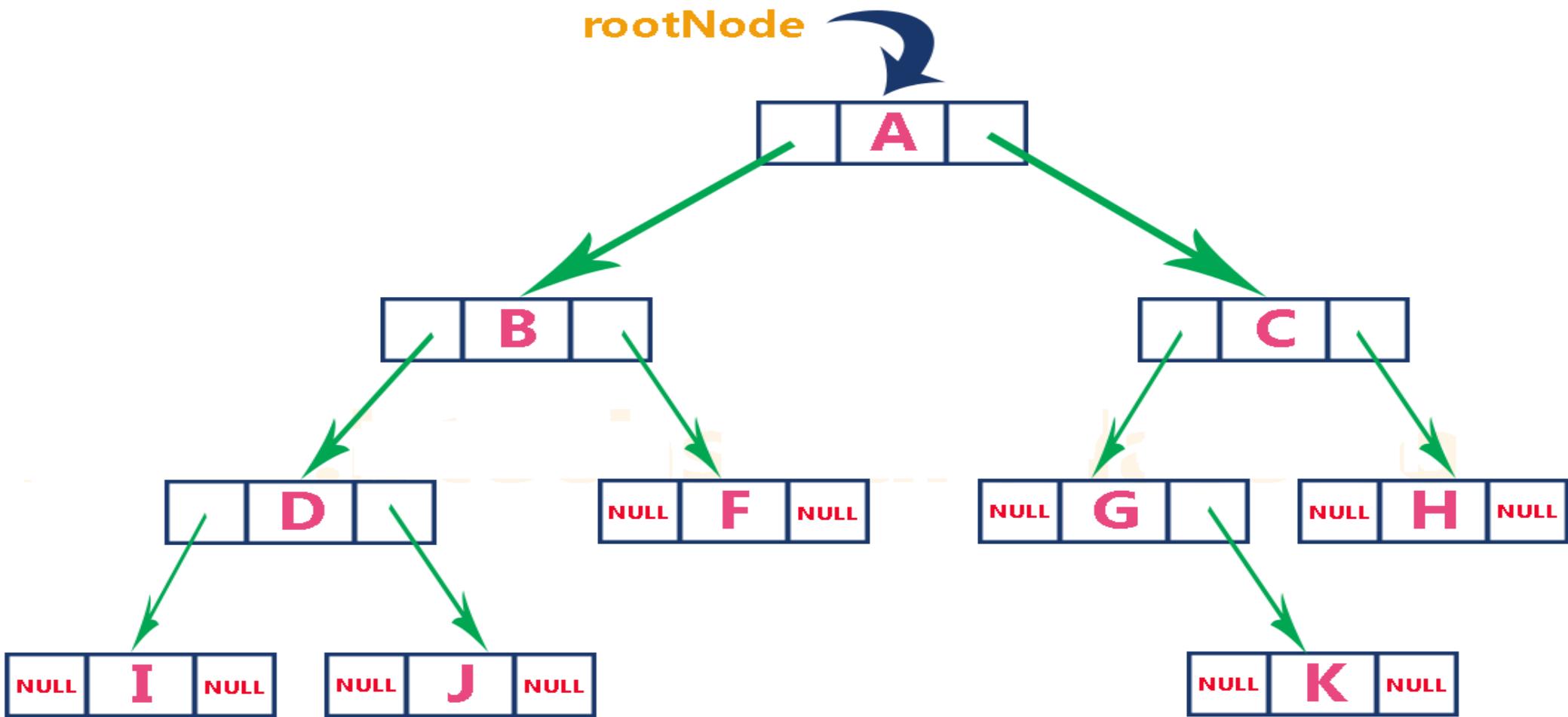
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Linked List Representation

We Use A Double Linked List To Represent A Binary Tree.

- In A Double Linked List, Every Node Consists Of Three Fields.
- First Field For Storing Left Child Address.
- Second For Storing Actual Data.
- Third For Storing Right Child Address.

Left Child Address	Data	Right Child Address
--------------------	------	---------------------

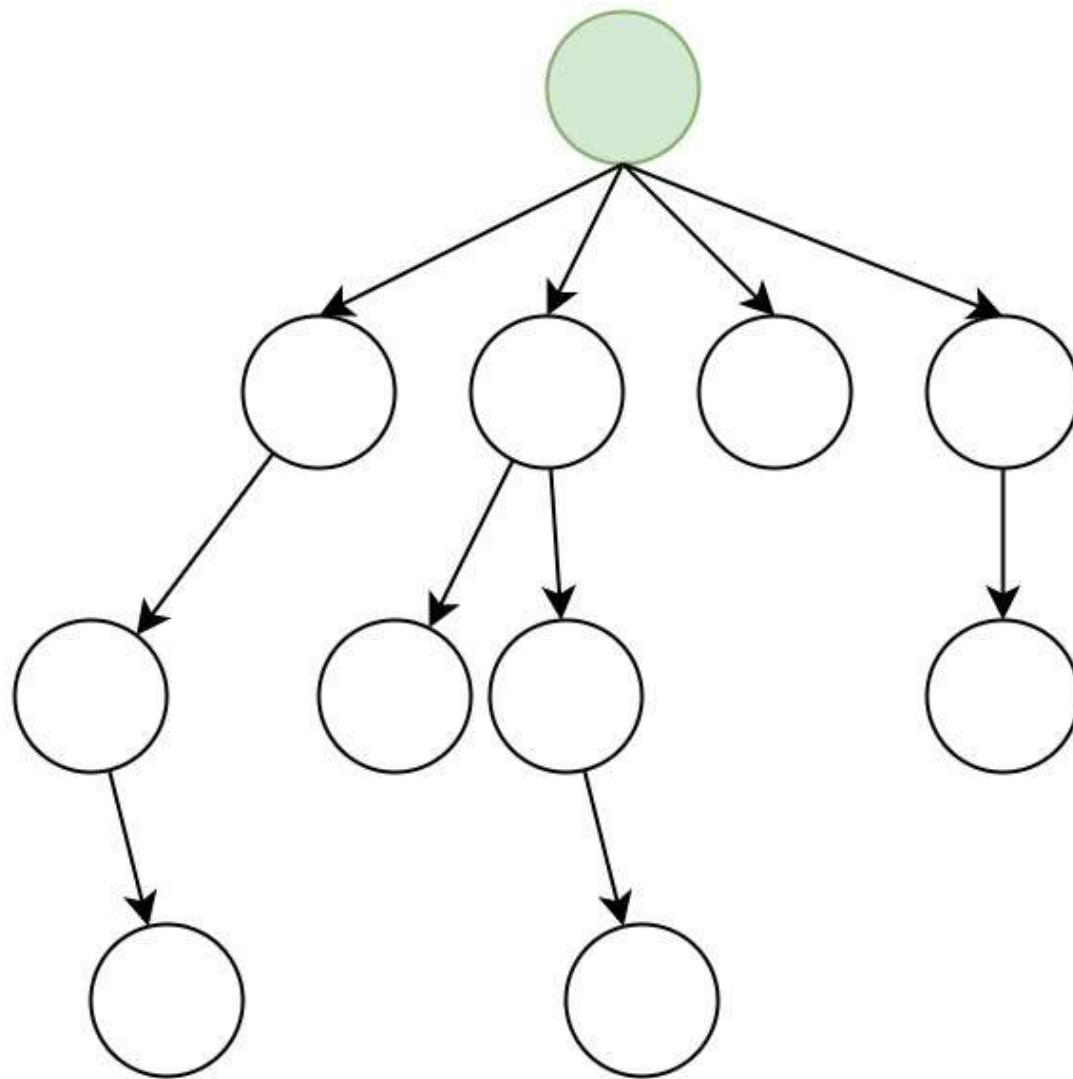


TYPES OF TREES

- 1. General trees/ forest trees**
- 2. Binary trees**
- 3. Expression trees**
- 4. Binary search trees**
- 5. Heap Trees**
- 6. Height Balanced Trees**
- 7. B. Trees**
- 8. B+ Trees**
- 9. Red Black Trees.**

General trees/ forest trees

- ✓ General tree is a tree in which each node can have either zero or many child nodes. It can not be empty.
- ✓ In general tree, there is no limitation on the degree of a node. The topmost node of a general tree is called the root node.
- ✓ There are many sub trees in a general tree.
- ✓ The sub tree of a general tree is unordered because the nodes of the general tree can not be ordered according to specific criteria

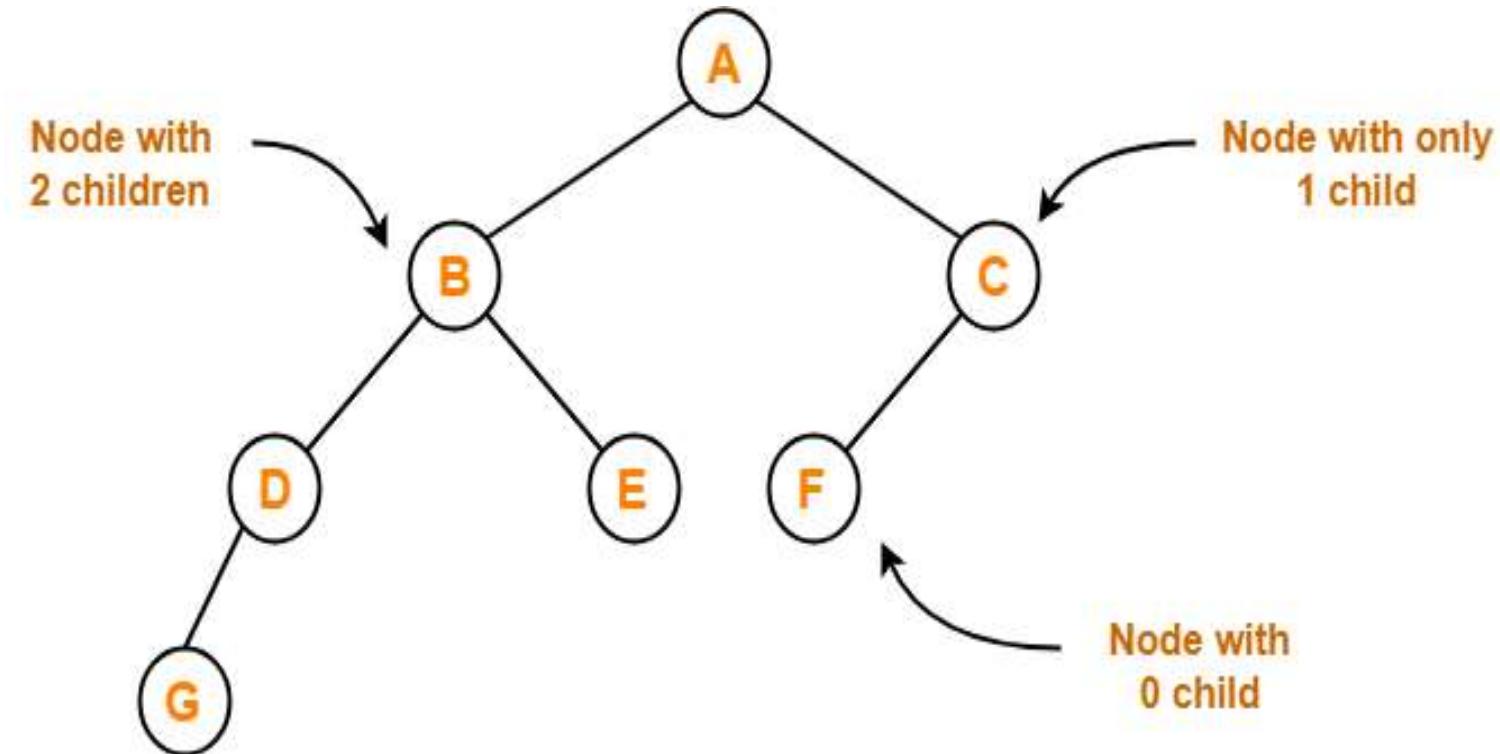


General Tree

Binary Tree

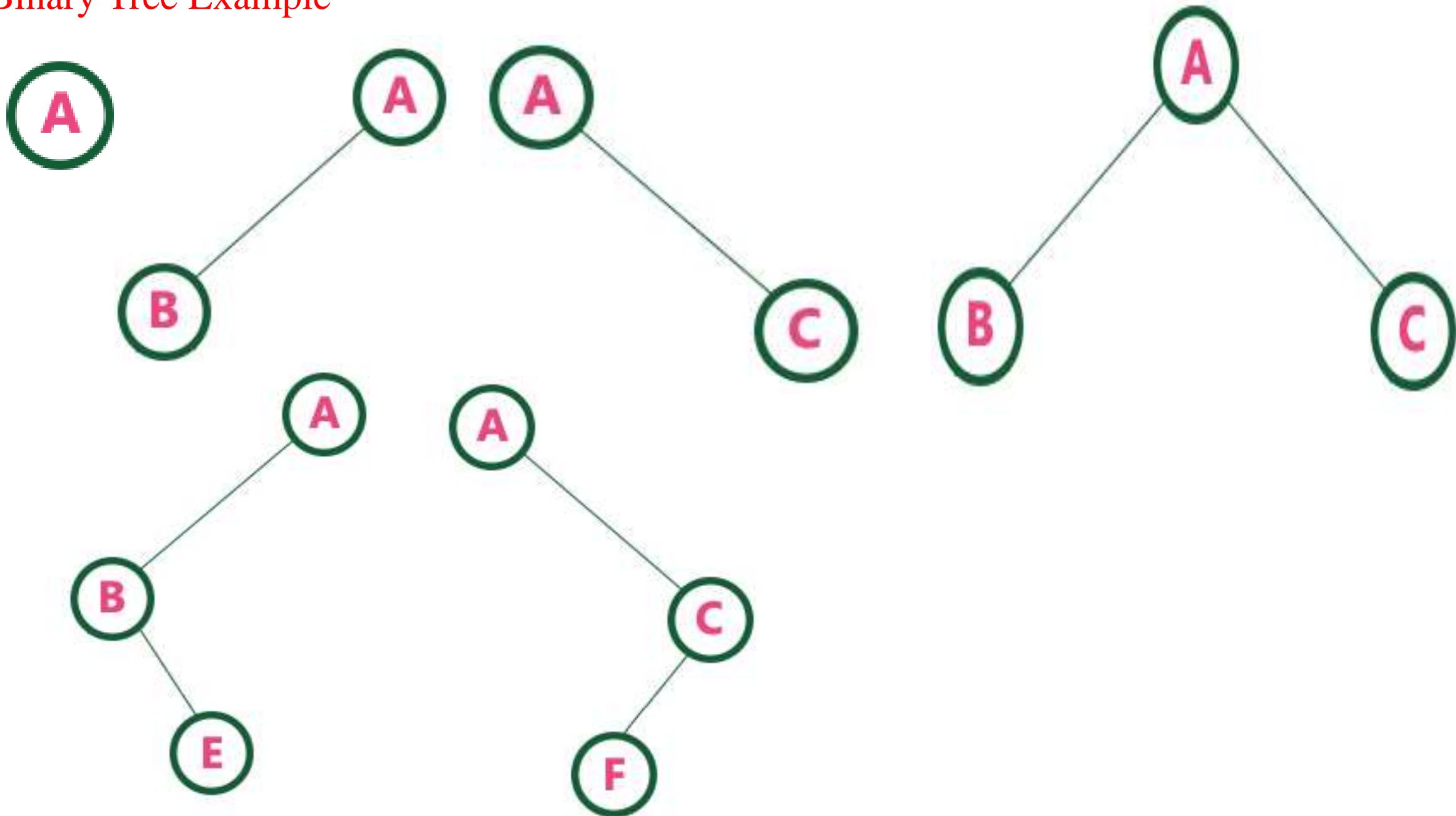
Binary Trees

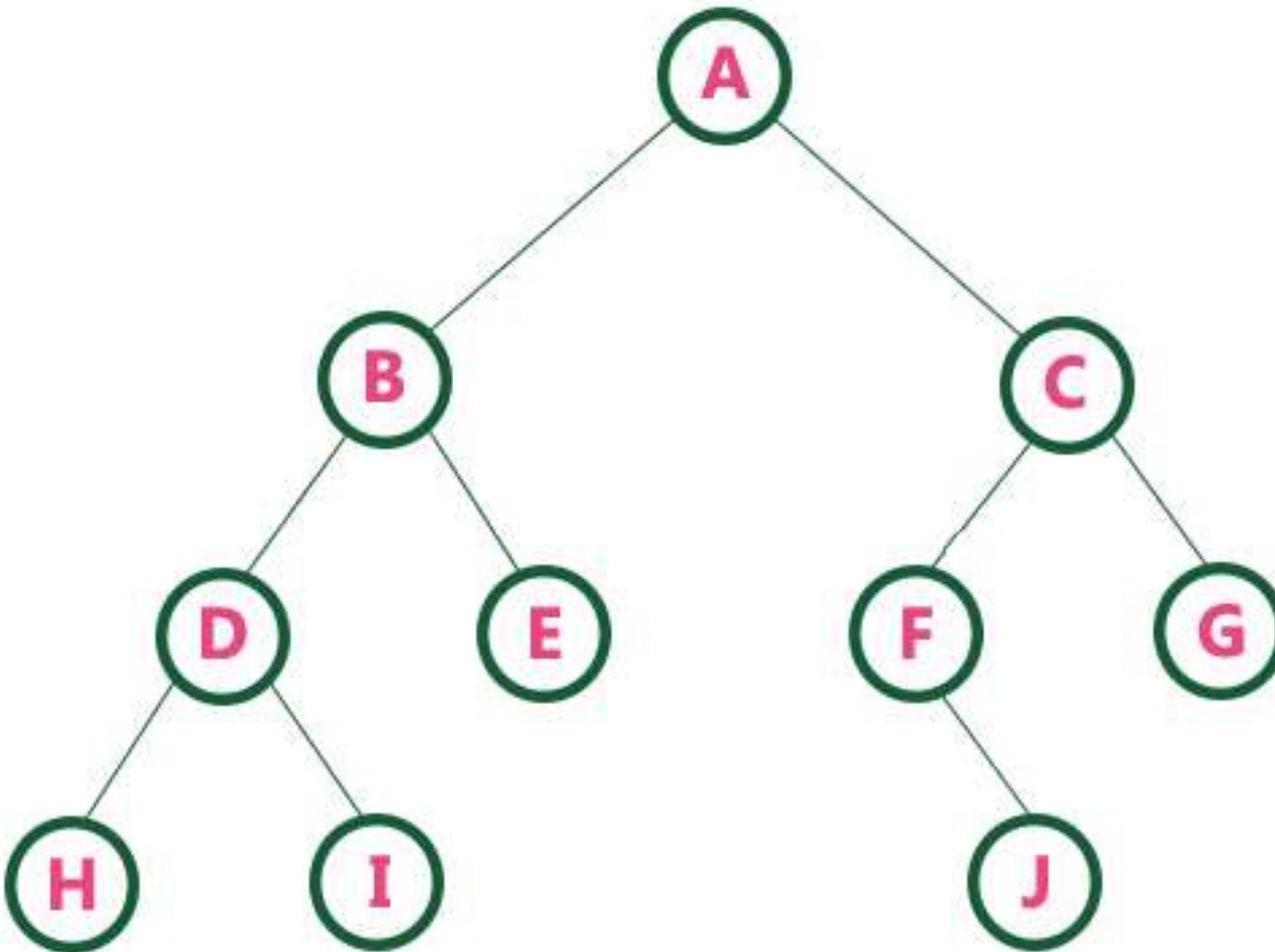
- ✓ Binary tree is a special tree data structure in which each node can have at most 2 children.
- ✓ Thus, in a binary tree, each node has either 0 child or 1 child or 2 children.



Binary Tree Example

Binary Tree Example





Finally Binary Tree

Binary Tree Properties

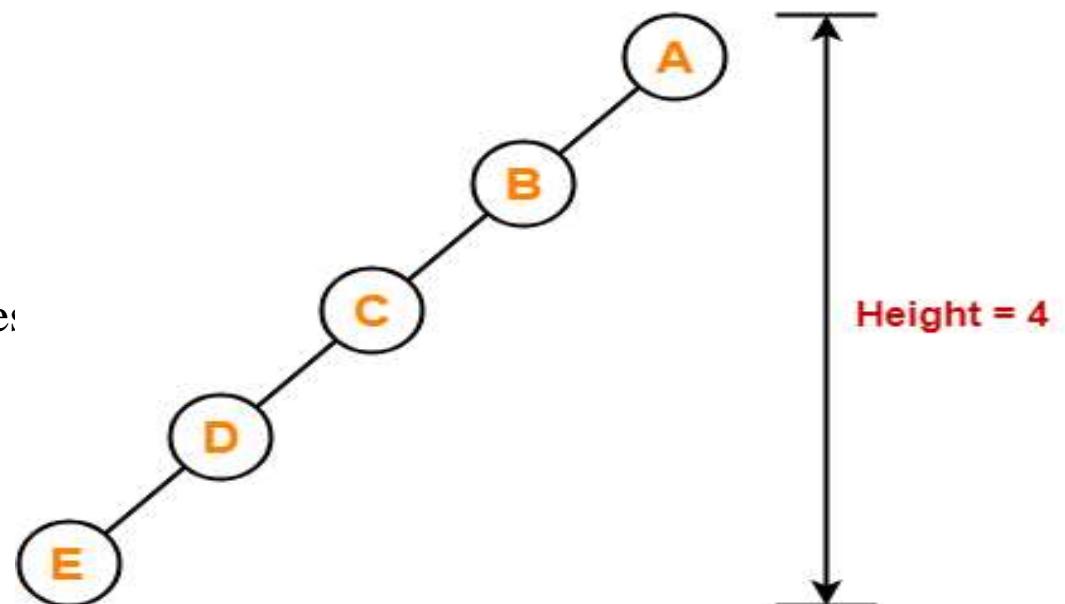
1. Minimum number of nodes in a binary tree of height $H = H + 1$
2. Maximum number of nodes in a binary tree of height $H = 2^H - 1$
3. Total Number of leaf nodes in a Binary Tree = Total Number of nodes with 2 children + 1
4. Maximum number of nodes at any level 'L' in a binary tree = 2^L

Minimum number of nodes in a binary tree of height $H = H + 1$

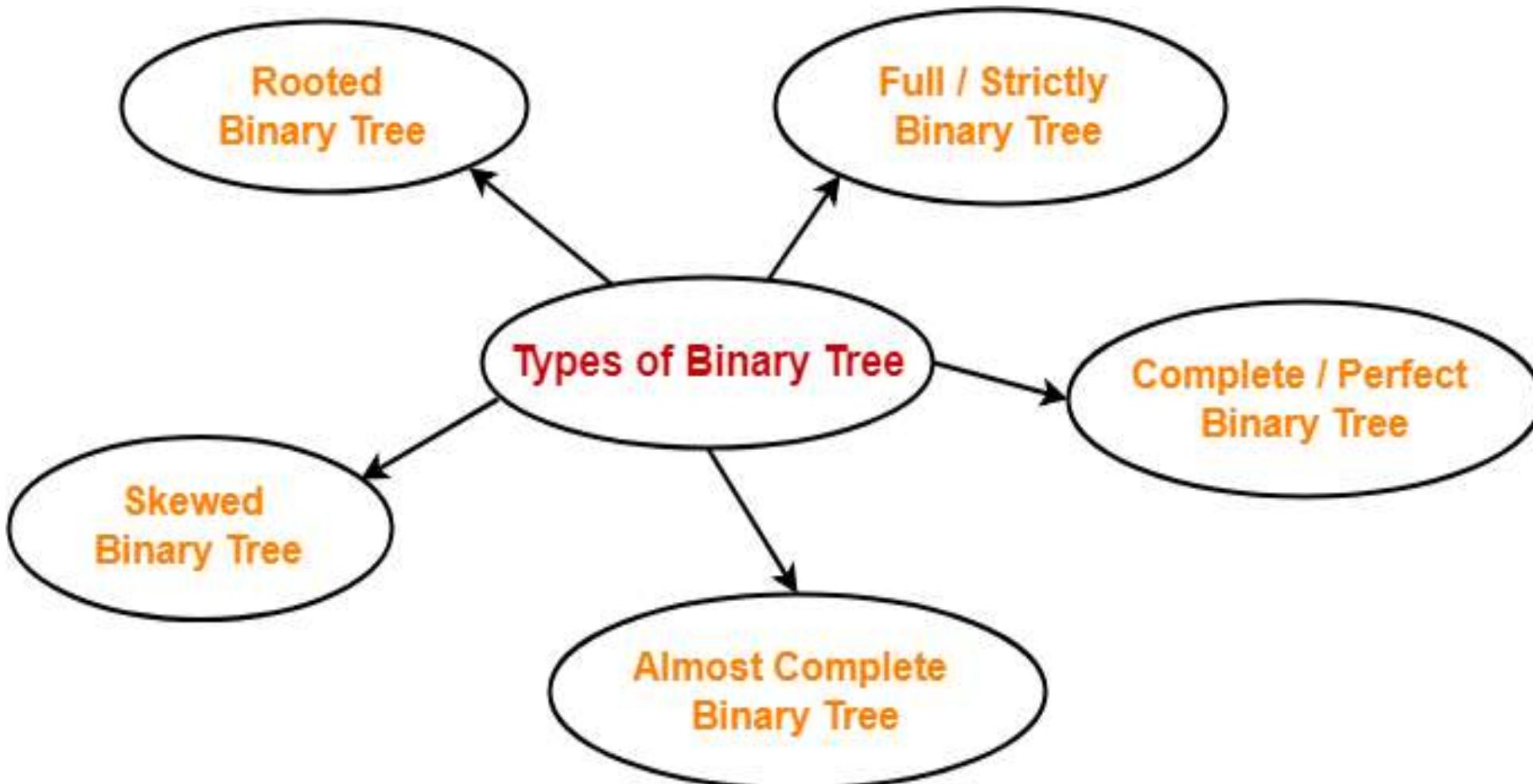
Where ' H ' is the height of the tree

Example

To construct a binary tree of height = 4, we need at least $4 + 1 = 5$ nodes:



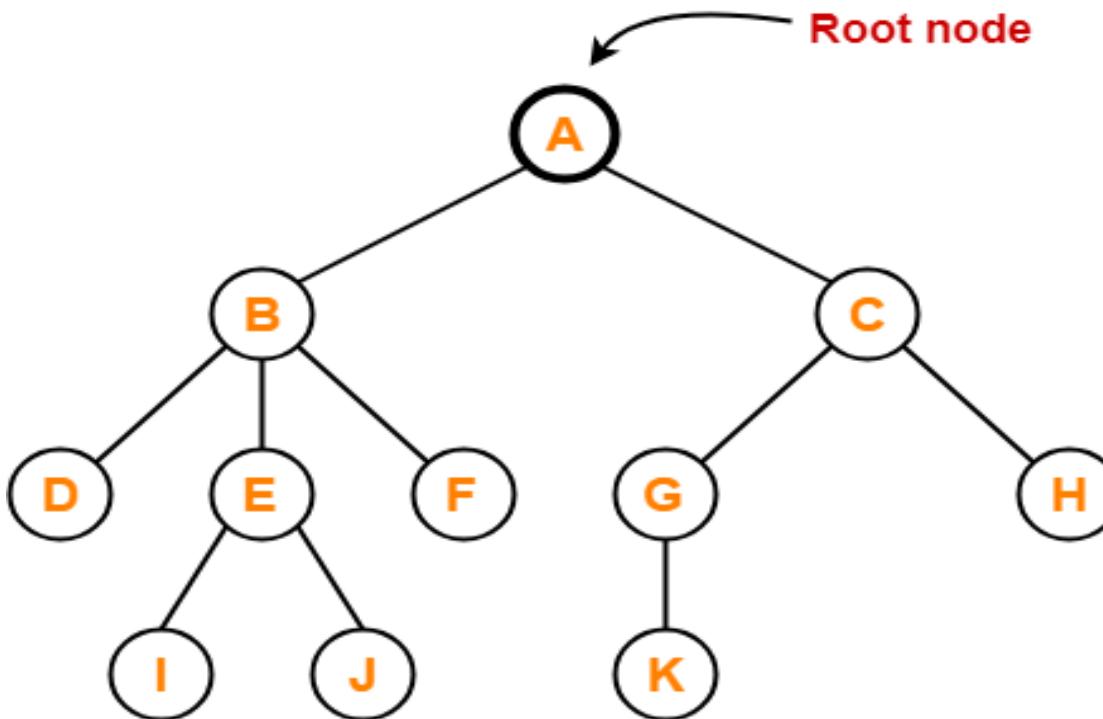
Types of Binary Trees



Rooted Binary Tree

A rooted binary tree is a binary tree that satisfies the following 2 properties

- ✓ It has a root node.
- ✓ Each node has at most 2 children.

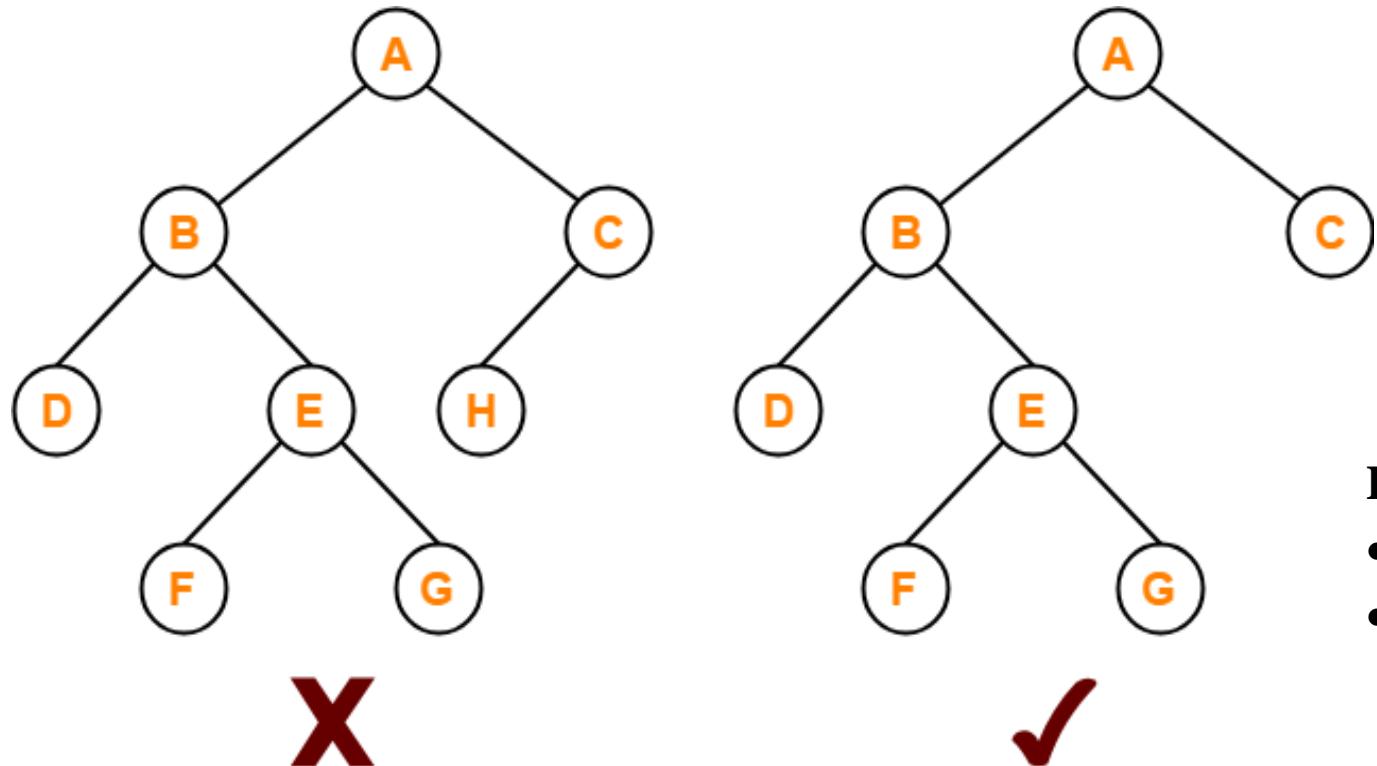


Full / Strictly Binary Tree

A binary tree in which every node has either 0 or 2 children is called as a full binary tree.

- Full binary tree is also called as strictly binary tree.

Every node must have two children except the leaf nodes



Here

- First binary tree is not a full binary tree.
- This is because node C has only 1 child.

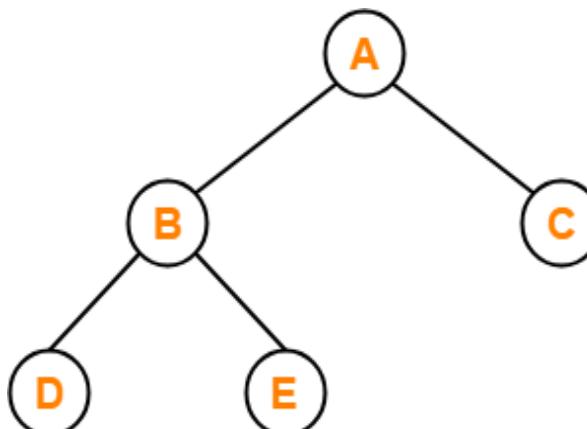
Complete / Perfect Binary Tree

A complete binary tree is a binary tree that satisfies the following 2 properties

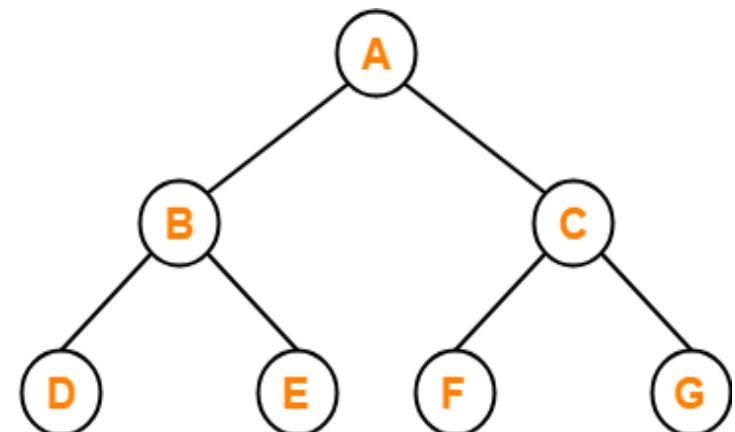
- ✓ Every internal node has exactly 2 children.
- ✓ All the leaf nodes are at the same level.
- ✓ Each level there must be 2^L nodes where L is level

Complete binary tree is also called as Perfect binary tree.

$2^0 = 1$ node in 0th level



$2^1 = 2$ nodes in 1st level



$2^2 = 4$ nodes in 2nd level

$2^3 = 8$ nodes in 3rd level

Here

X

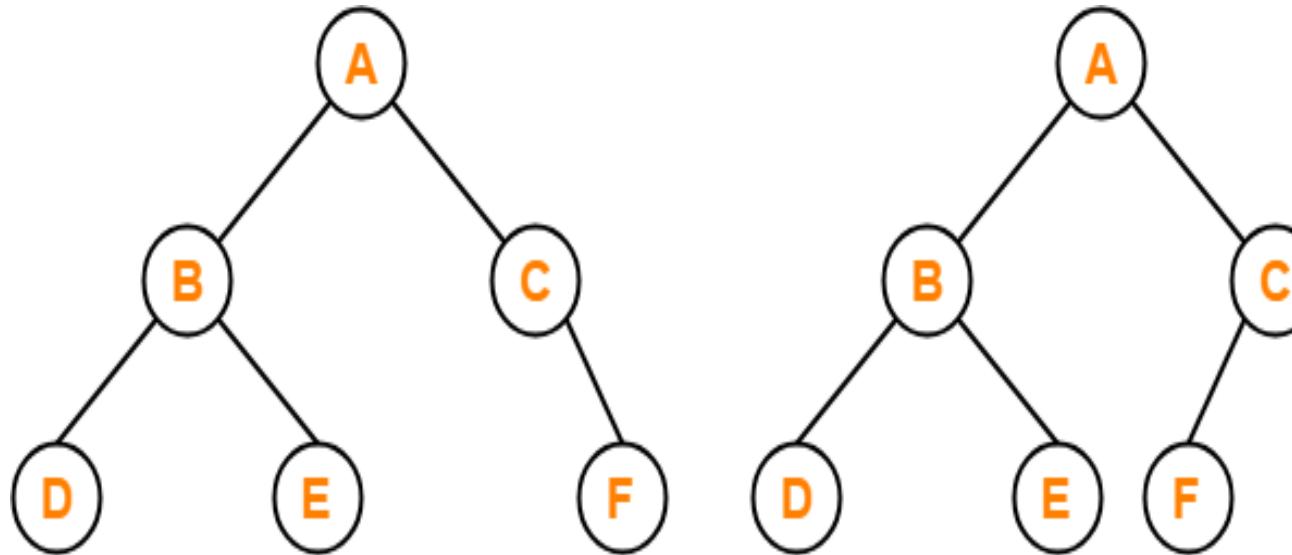
✓

- First binary tree is not a complete binary tree.
- This is because all the leaf nodes are not at the same level.

Almost Complete Binary Tree

An almost complete binary tree is a binary tree that satisfies the following 2 properties

- ✓ Every node must have two children in all the levels. Except in last level.
- ✓ The last level must be strictly filled from left to right.

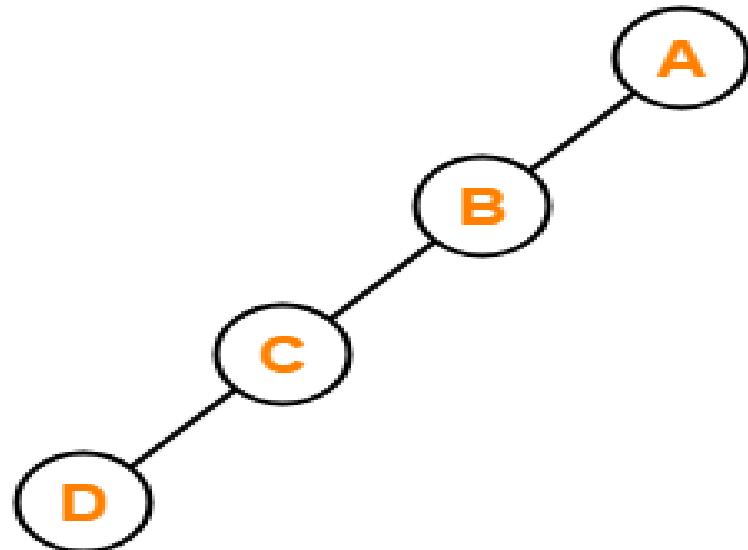


X

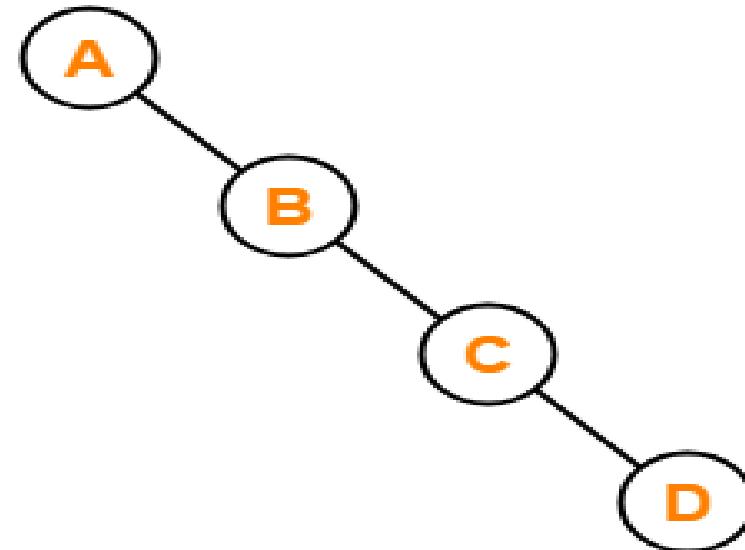
✓

Skewed Binary Tree

- ✓ Every node should have either left or right children only.
 - ✓ The remaining node has no child.
 - ✓ There are two types of Skewed binary trees i) left skewed ii) right skewed
- OR
- ✓ A skewed binary tree is a binary tree of n nodes such that its depth is $(n-1)$.



Left Skewed Binary Tree

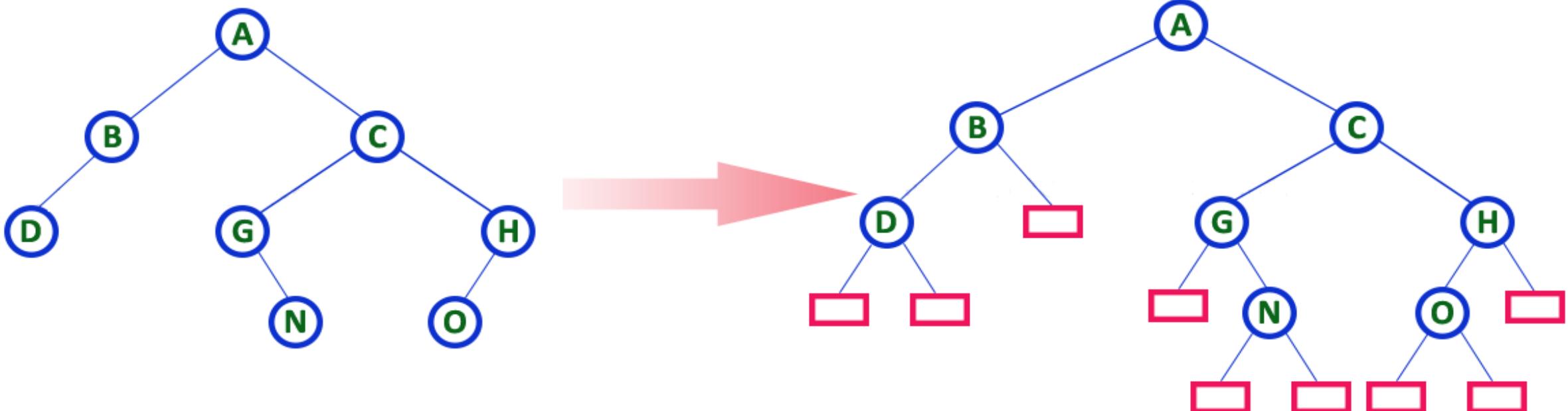


Right Skewed Binary Tree

Extended Binary Trees

In an extended binary tree, nodes having two children are called internal nodes and nodes having no children are called external nodes. The internal nodes are represented using circles and the external nodes are represented using squares.

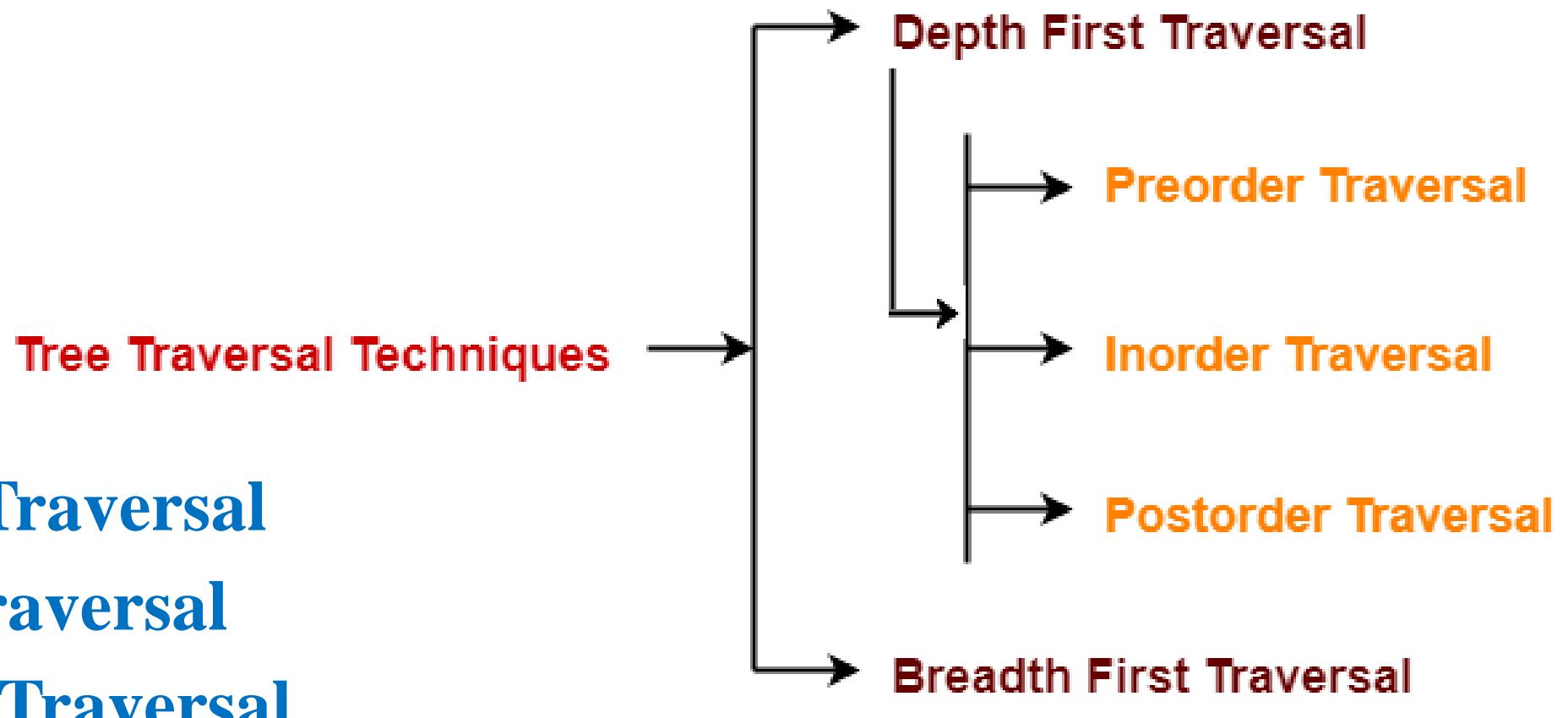
To convert a binary tree into an extended tree, every empty sub-tree is replaced by a new node. The original nodes in the tree are the internal nodes, and the new nodes added are called the external nodes.



Binary Tree Traversal

Binary Tree Traversals

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree.



NLR

Pre-order Traversal

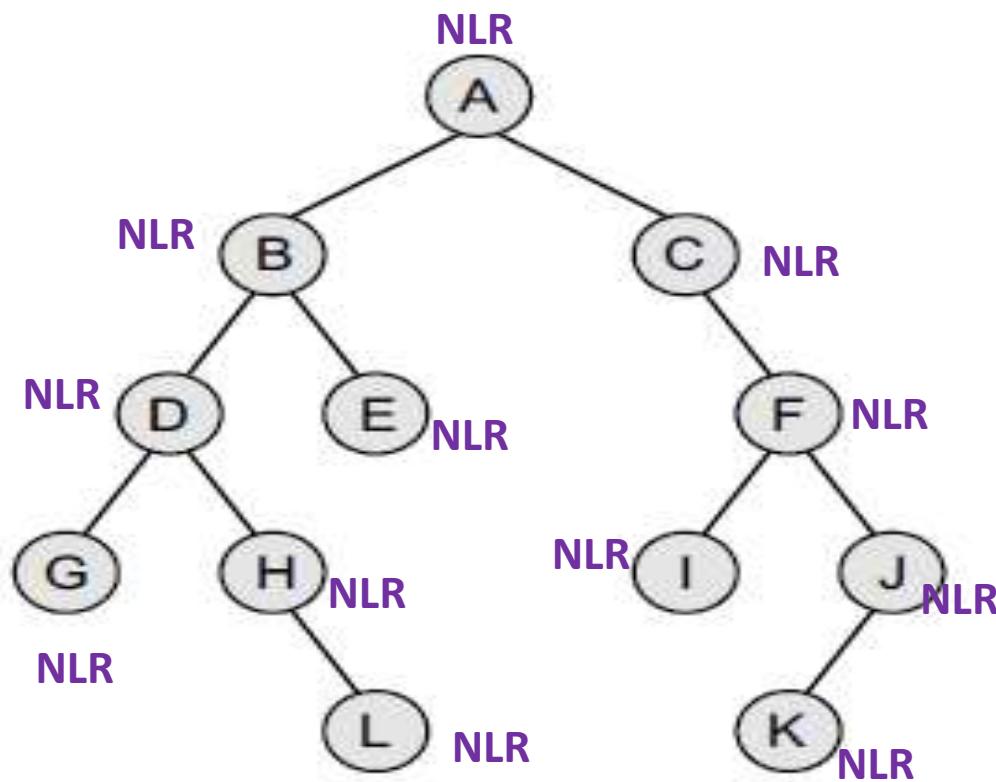
To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node.

1. Visiting the root node,
2. Traversing the left sub-tree, and finally
3. Traversing the right sub-tree.

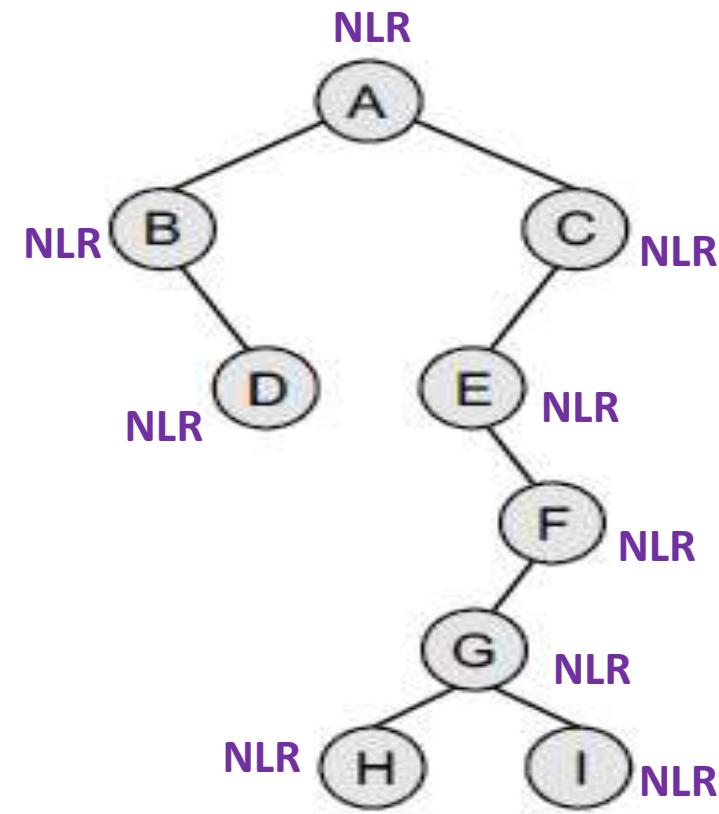
Algorithm for Pre-order

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL  
Step 2: Write TREE->DATA  
Step 3: PREORDER(TREE->LEFT)  
Step 4: PREORDER(TREE->RIGHT)  
        [END OF LOOP]  
Step 5: END
```

EXAMPLE PRE-ORDER



(a)



(b)

(a) TRAVERSAL ORDER: A, B, D, G, H, L, E, C, F, I, J, and K

(b) TRAVERSAL ORDER: A, B, D, C, D, E, F, G, H, and I

NLR

In-order Traversal

LNR

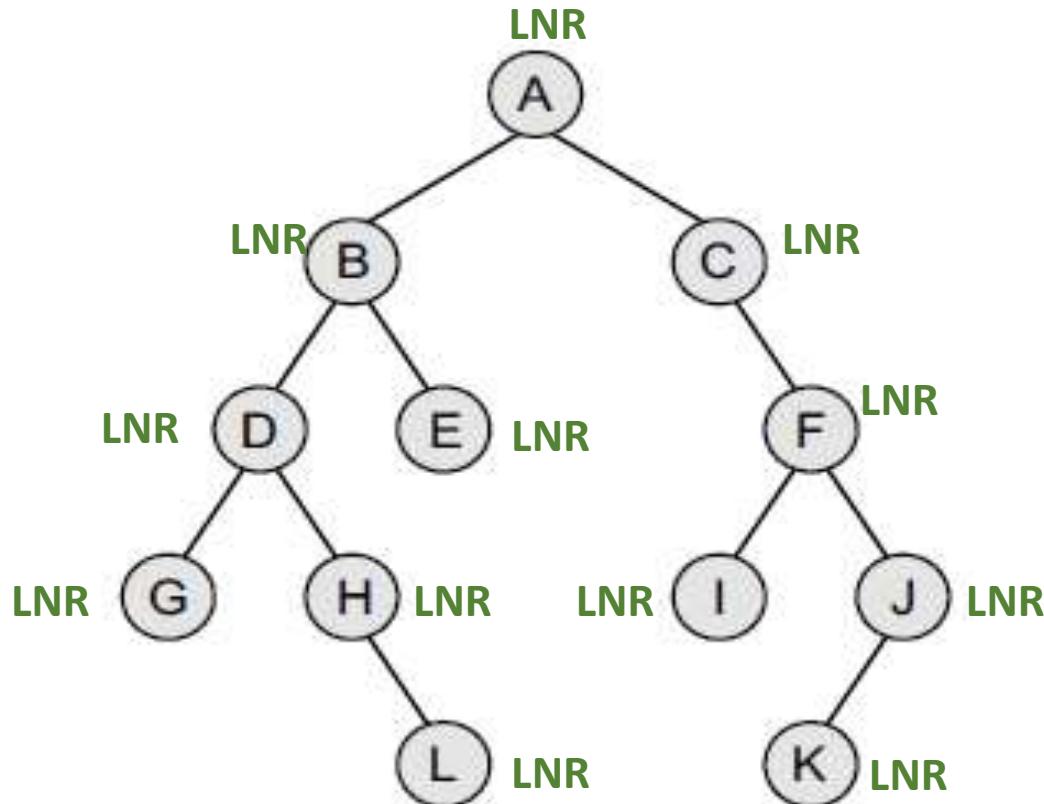
To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

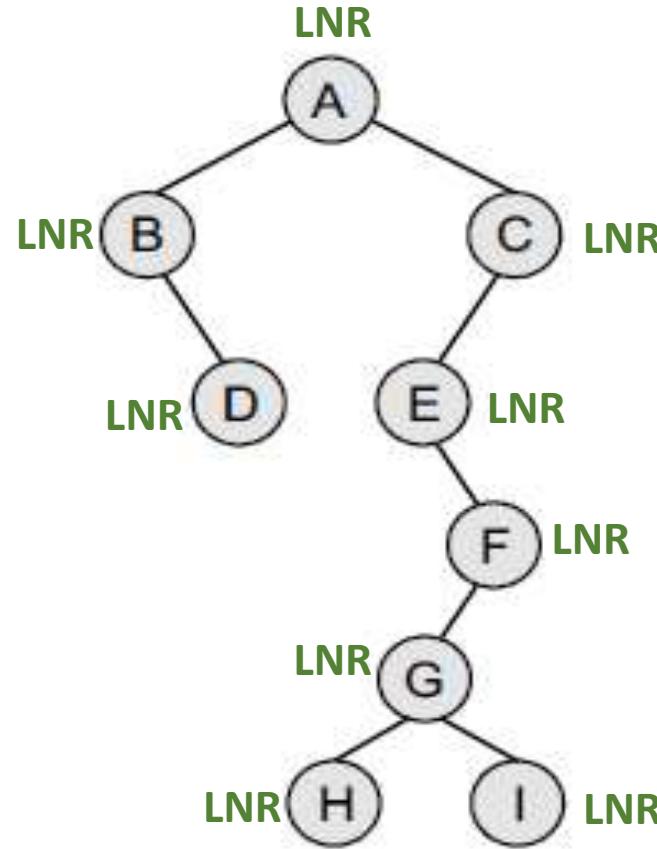
Algorithm for In-order

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL  
Step 2: INORDER(TREE → LEFT)  
Step 3: Write TREE → DATA  
Step 4: INORDER(TREE → RIGHT)  
        [END OF LOOP]  
Step 5: END
```

EXAMPLE In-ORDER



(a)



(b)

LNR

- (a) TRAVERSAL ORDER: G, D, H, L, B, E, A, C, I, F, K, and J
(b) TRAVERSAL ORDER: B, D, A, E, H, G, I, F, and C

Post-order Traversal

LRN

To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node.

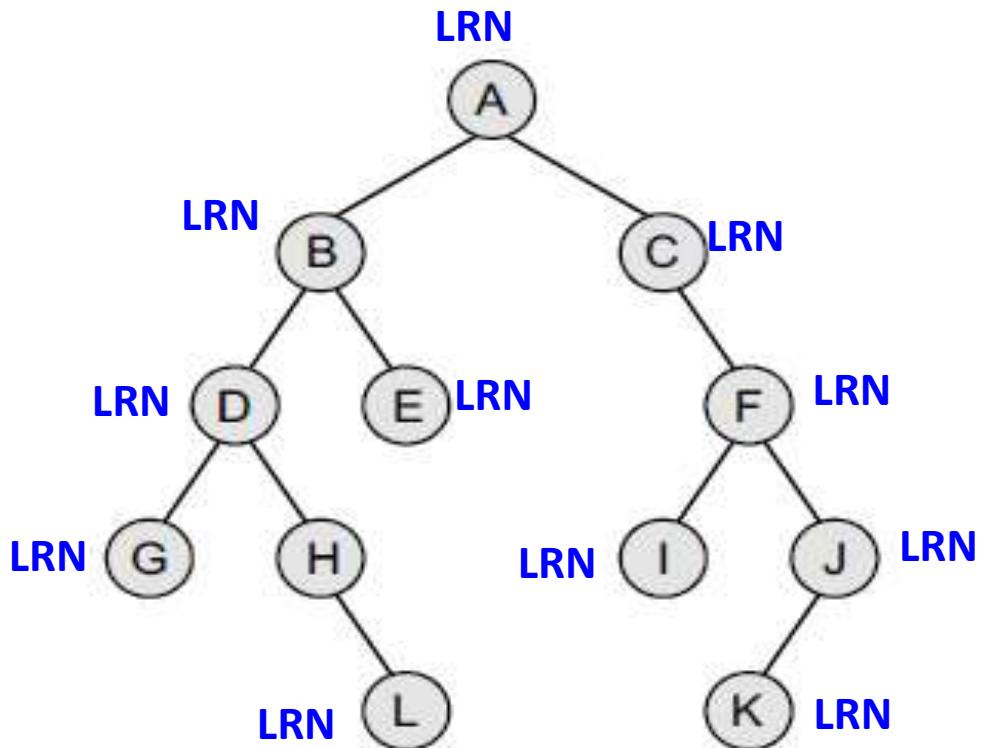
1. Traversing the left sub-tree,
2. Traversing the right sub-tree, and finally
3. Visiting the root node.

Algorithm for Post-order

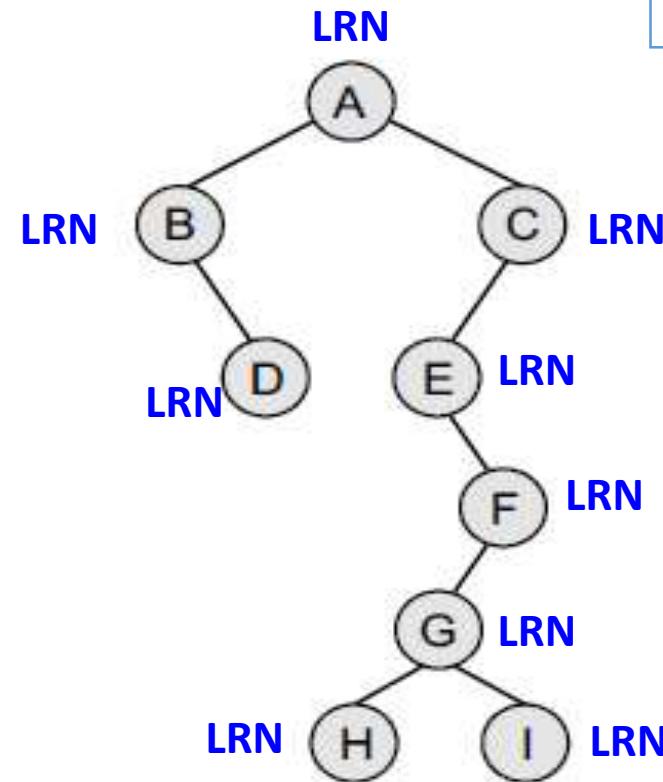
```
Step 1: Repeat Steps 2 to 4 while TREE != NULL  
Step 2: POSTORDER(TREE → LEFT)  
Step 3: POSTORDER(TREE → RIGHT)  
Step 4: Write TREE → DATA  
        [END OF LOOP]  
Step 5: END
```

EXAMPLE Post-ORDER

LRN



(a)



(b)

(a) TRAVERSAL ORDER: G, L, H, D, E, B, I, K, J, F, C, and A

(b) TRAVERSAL ORDER: D, B, H, I, G, F, E, C, and A

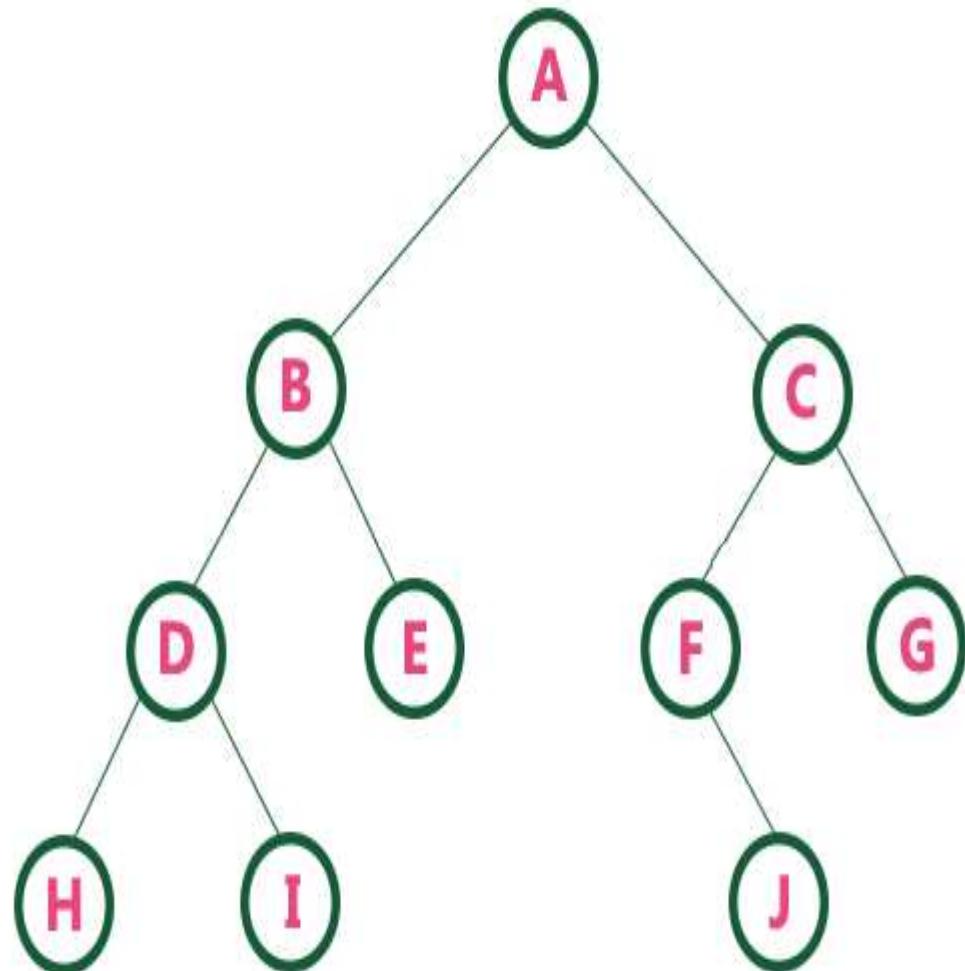
Construction of Binary tree with In-order & pre-order

In - Order :- LEFT NODE RIGHT

Pre - Order :- NODE LEFT RIGHT

Pre Order for the Tree :- A B D H I E C F J G

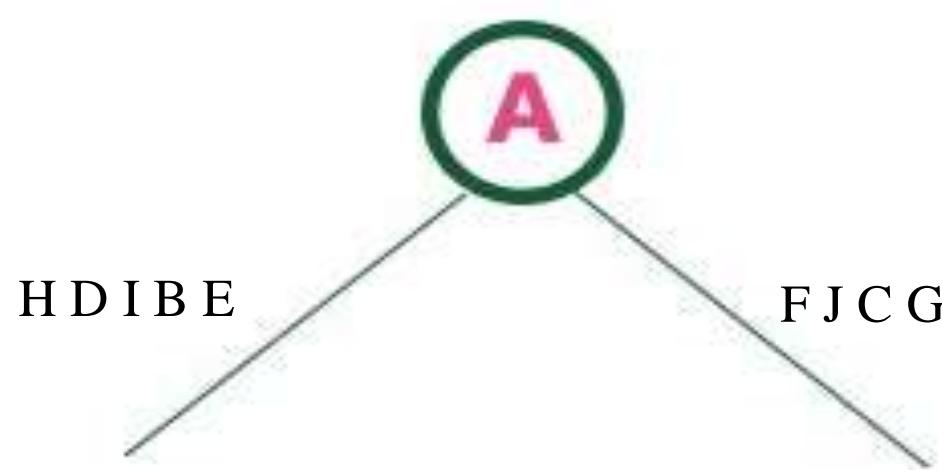
In Order for the Tree :- H D I B E A F J C G



Now finding the ROOT from PRE ORDER
and
LEFT & RIGHT CHILD'S from IN ORDER

Pre Order for the Tree :- A B D H I E C F J G

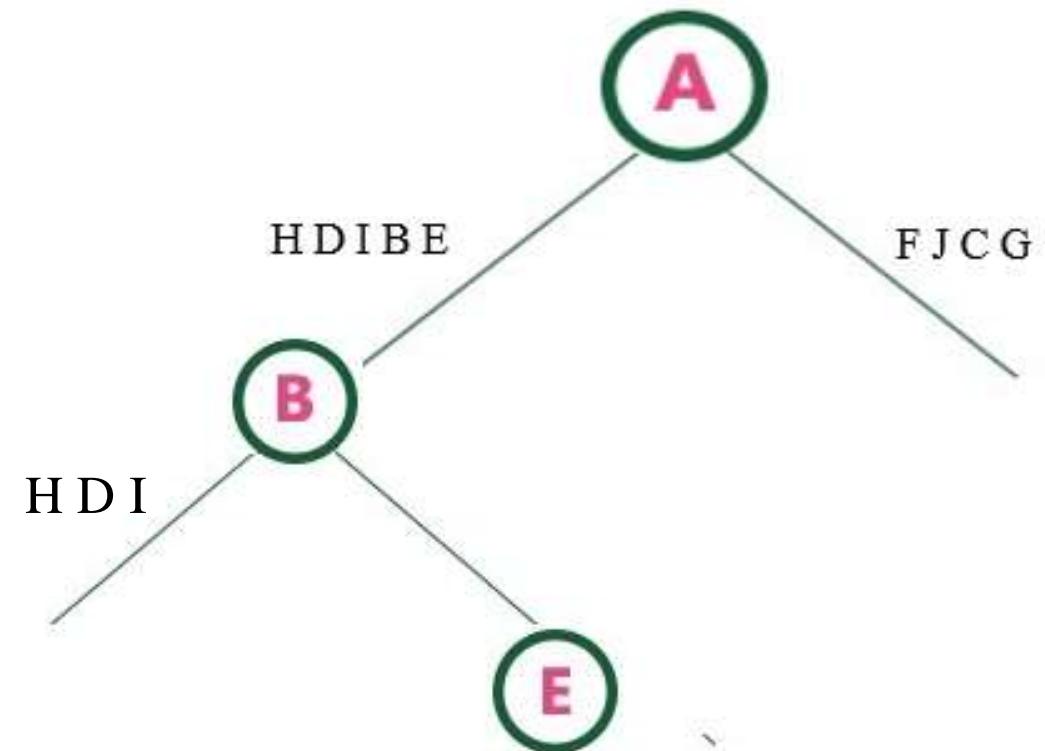
In Order for the Tree :- H D I B E A F J C G



Now finding the ROOT from PRE ORDER
and
LEFT & RIGHT CHILD'S from IN ORDER

Pre Order for the Tree :- A B D H I E C F J G

In Order for the Tree :- H D I B E A F J C G

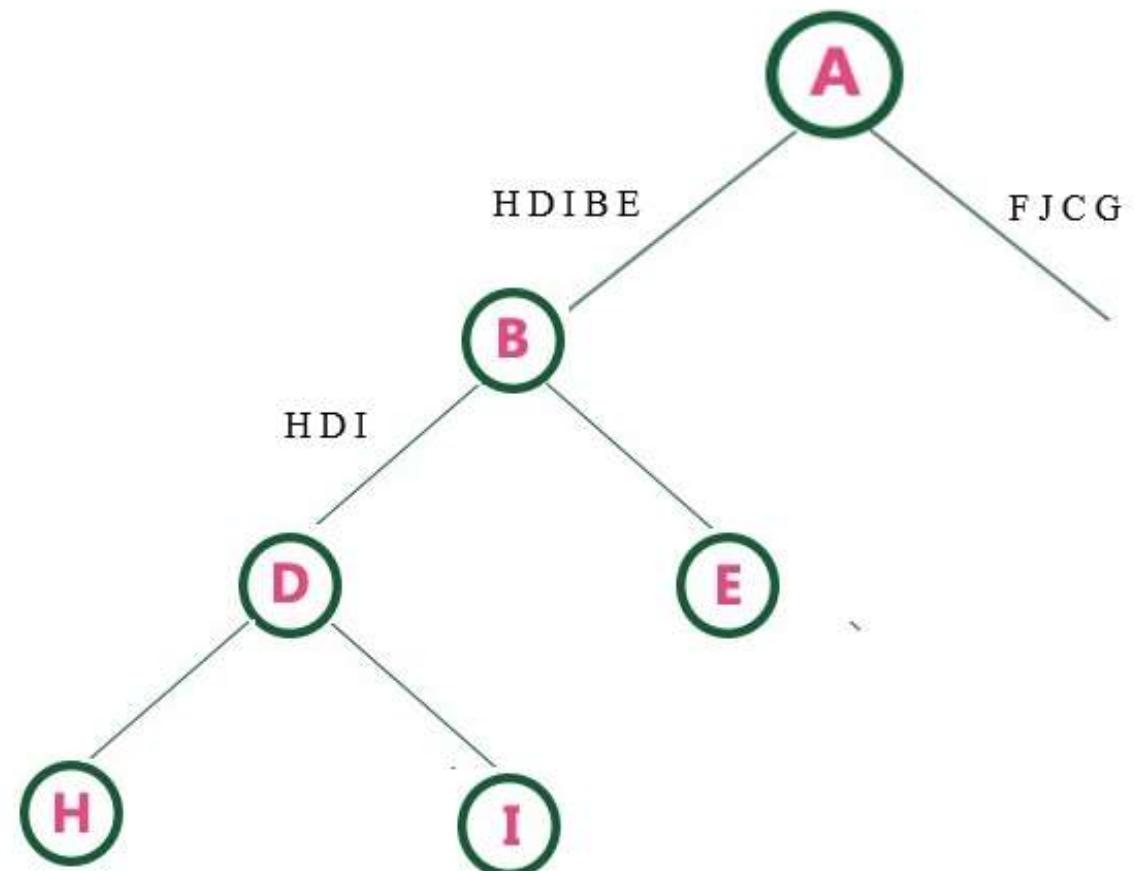


Now finding the ROOT from PRE ORDER
and
LEFT & RIGHT CHILD'S from IN ORDER

Pre Order for the Tree :- A B D H I E C F J G

In Order for the Tree :- H D I B E A F J C G

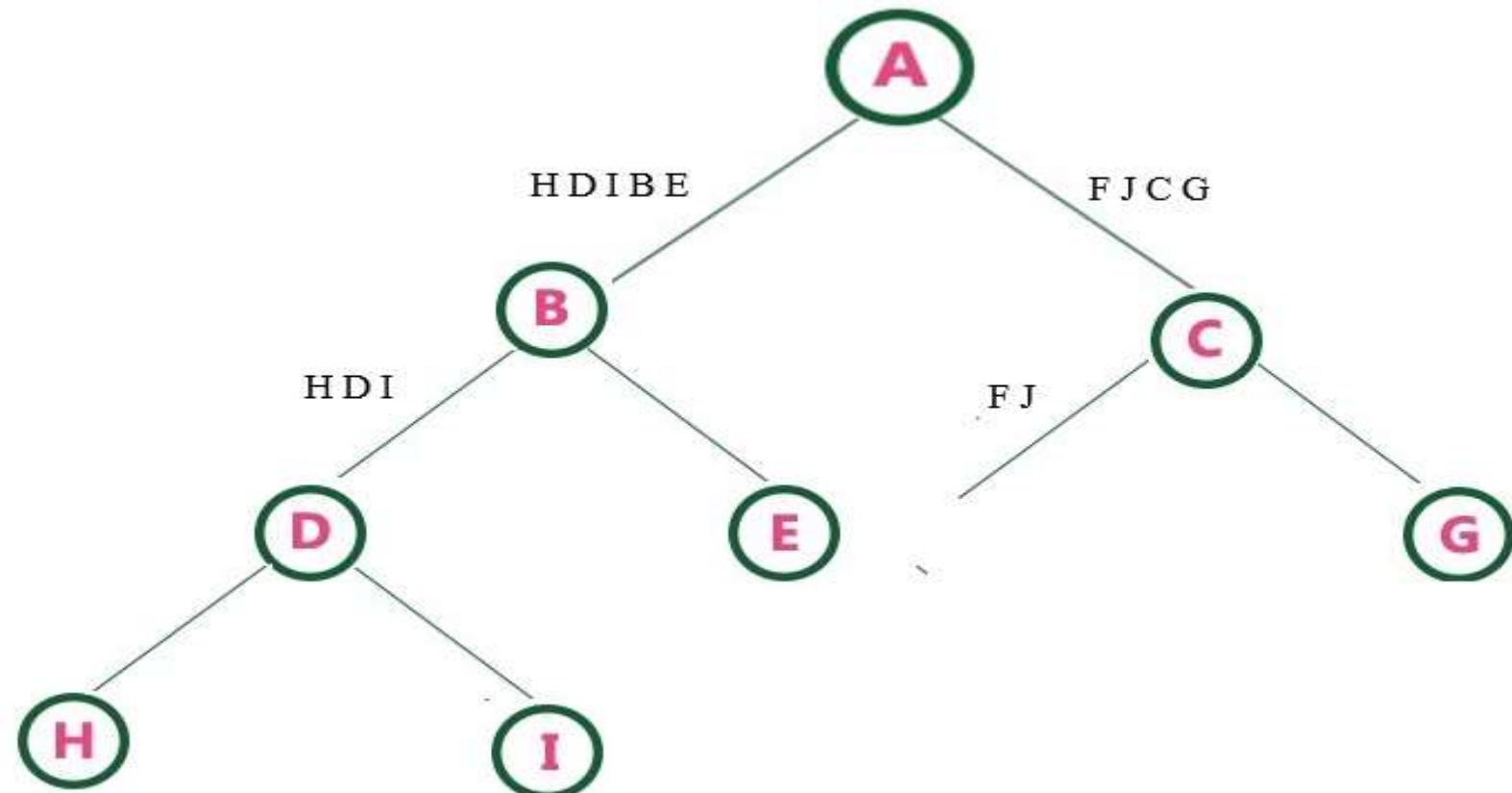
Now LEFT SUB TREE IS
COMPLETED



Now finding the ROOT from PRE ORDER
and
LEFT & RIGHT CHILD'S from IN ORDER

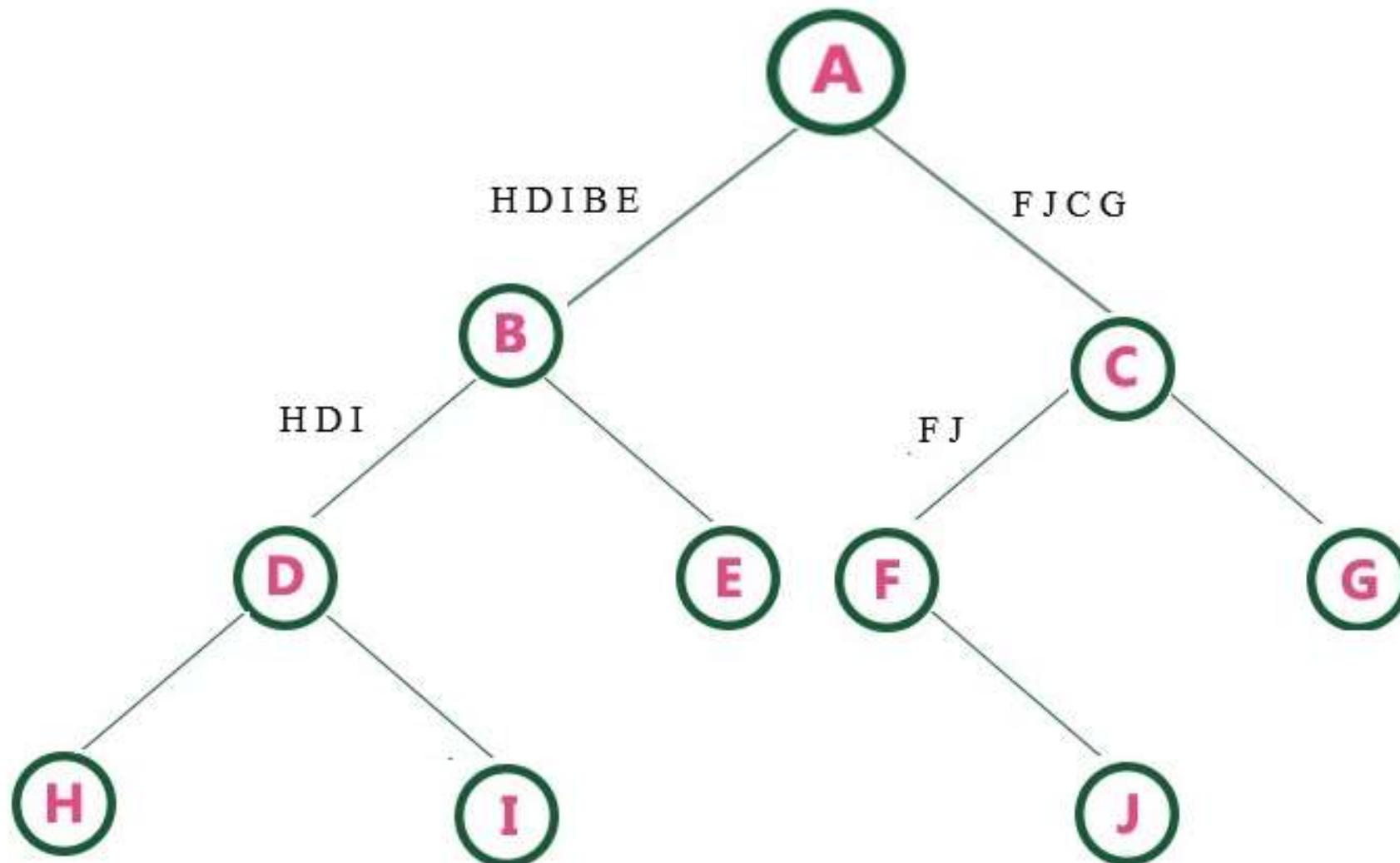
Pre Order for the Tree :- A B D H I E C F J G

In Order for the Tree :- H D I B E A F J C G



Now finding the ROOT from PRE ORDER
and
LEFT & RIGHT CHILD'S from IN

Pre Order for the Tree :- A B D H I E C F J G
In Order for the Tree :- H D I B E A F J C G



Construction of Binary tree

with

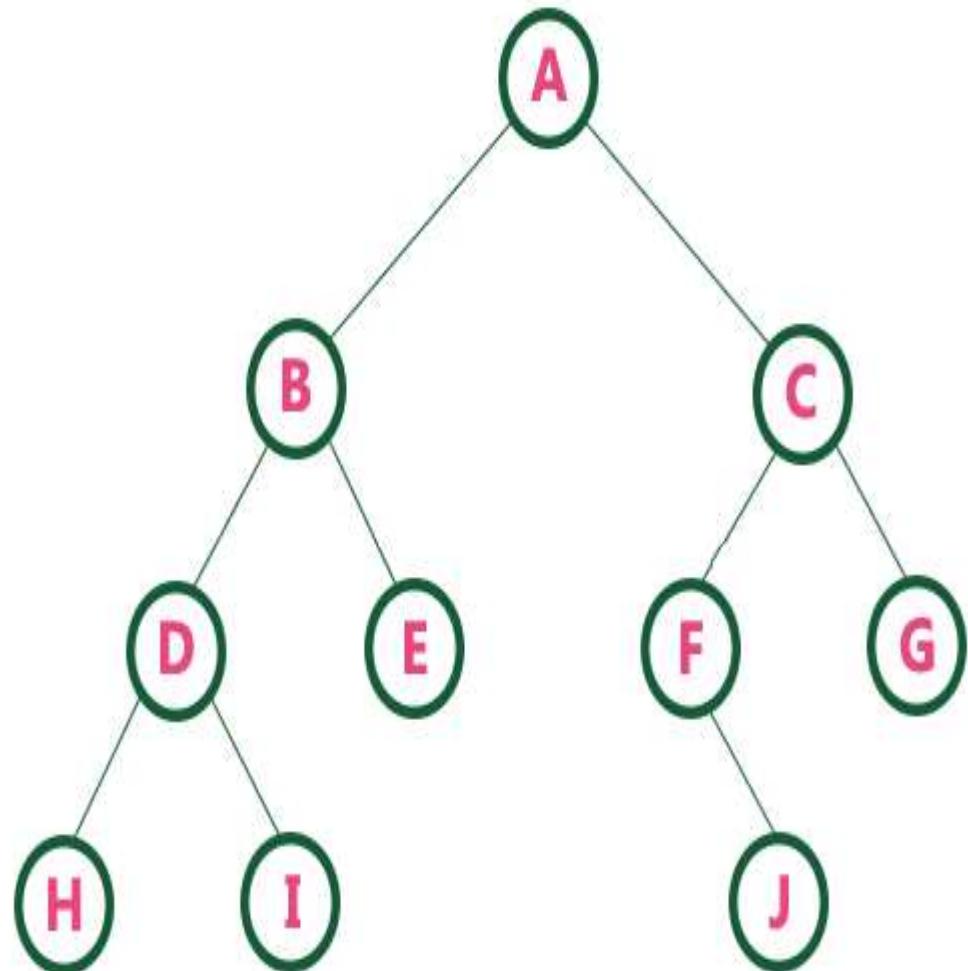
In-order & Post-order

In - Order :- LEFT NODE RIGHT

Post - Order :- LEFT RIGHT NODE

Post Order for the Tree :- H I D E B J F G C A

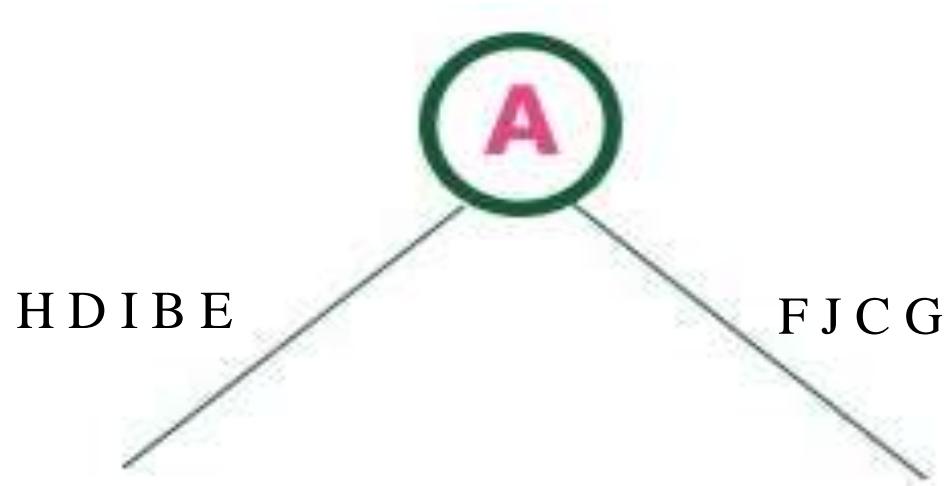
In Order for the Tree :- H D I B E A F J C G



Now finding the ROOT from POST ORDER
and
LEFT & RIGHT CHILD'S from IN ORDER

POST Order for the Tree :- H I D E B J F G C A

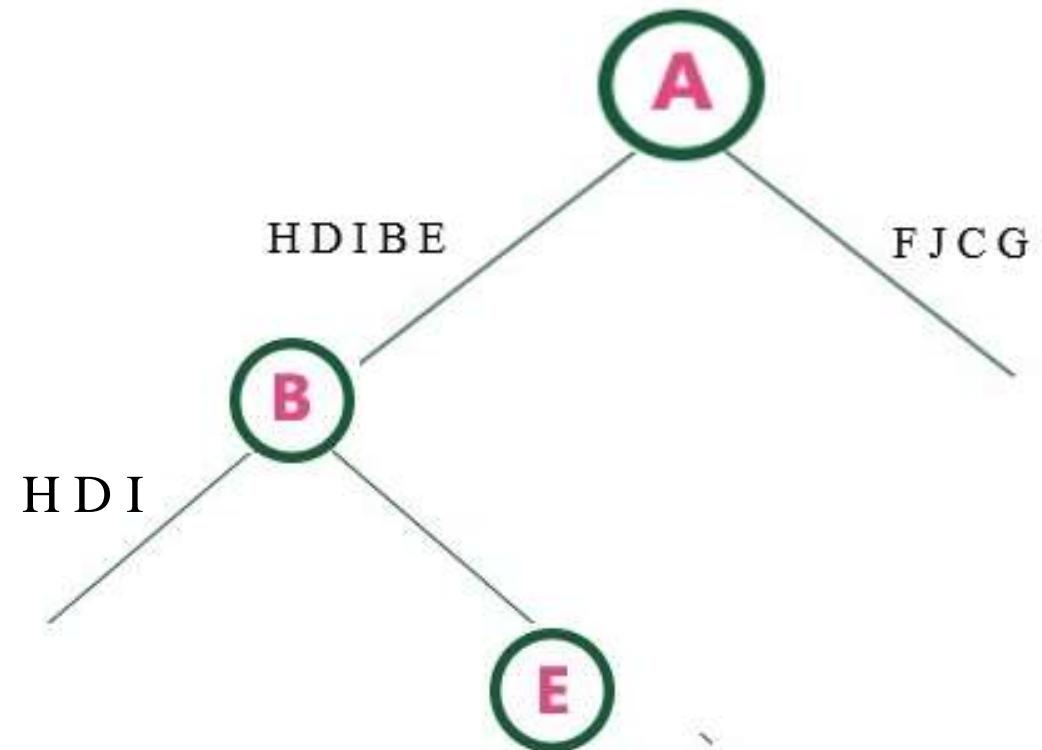
In Order for the Tree :- H D I B E A F J C G



Now finding the ROOT from POST ORDER
and
LEFT & RIGHT CHILD'S from IN ORDER

POST Order for the Tree :- H I D E B J F G C A

In Order for the Tree :- H D I B E A F J C G

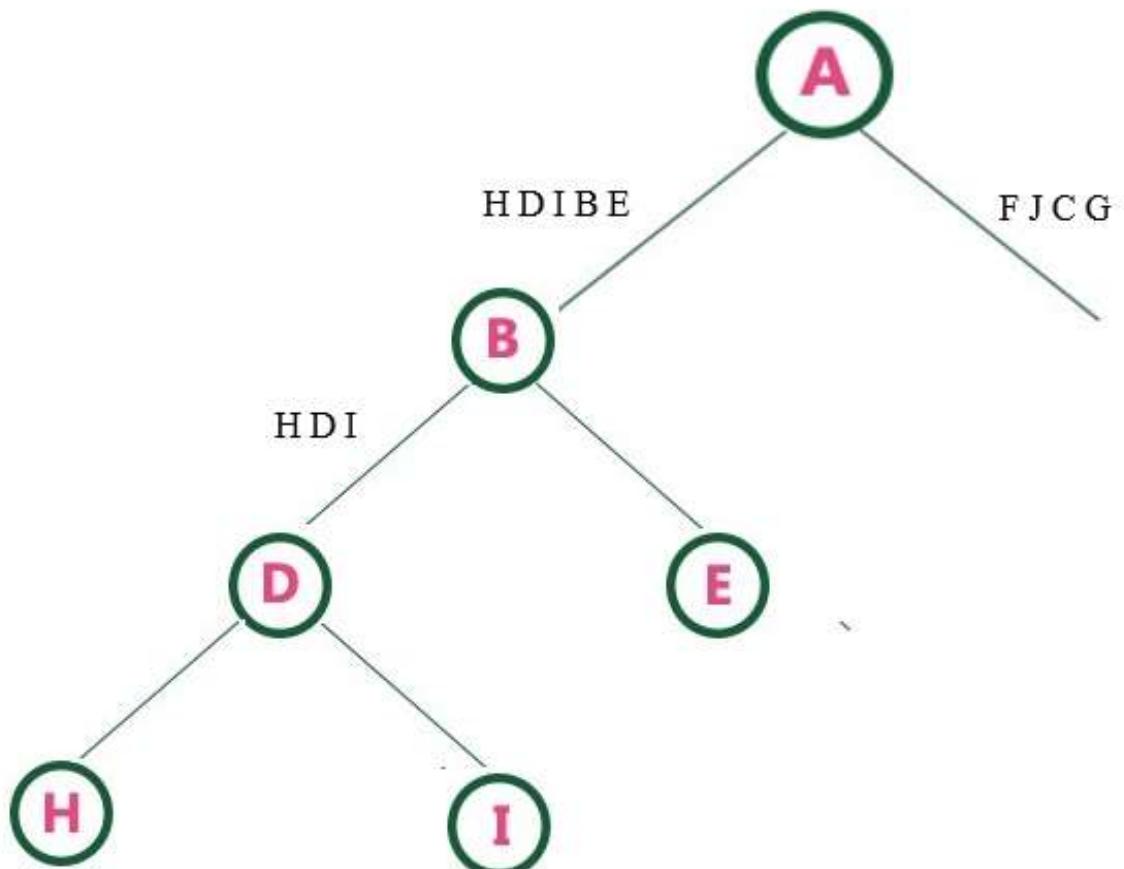


Now finding the ROOT from POST ORDER
and
LEFT & RIGHT CHILD'S from IN ORDER

POST Order for the Tree :- H D E B J F G C A

In Order for the Tree :- H D I B E A F J C G

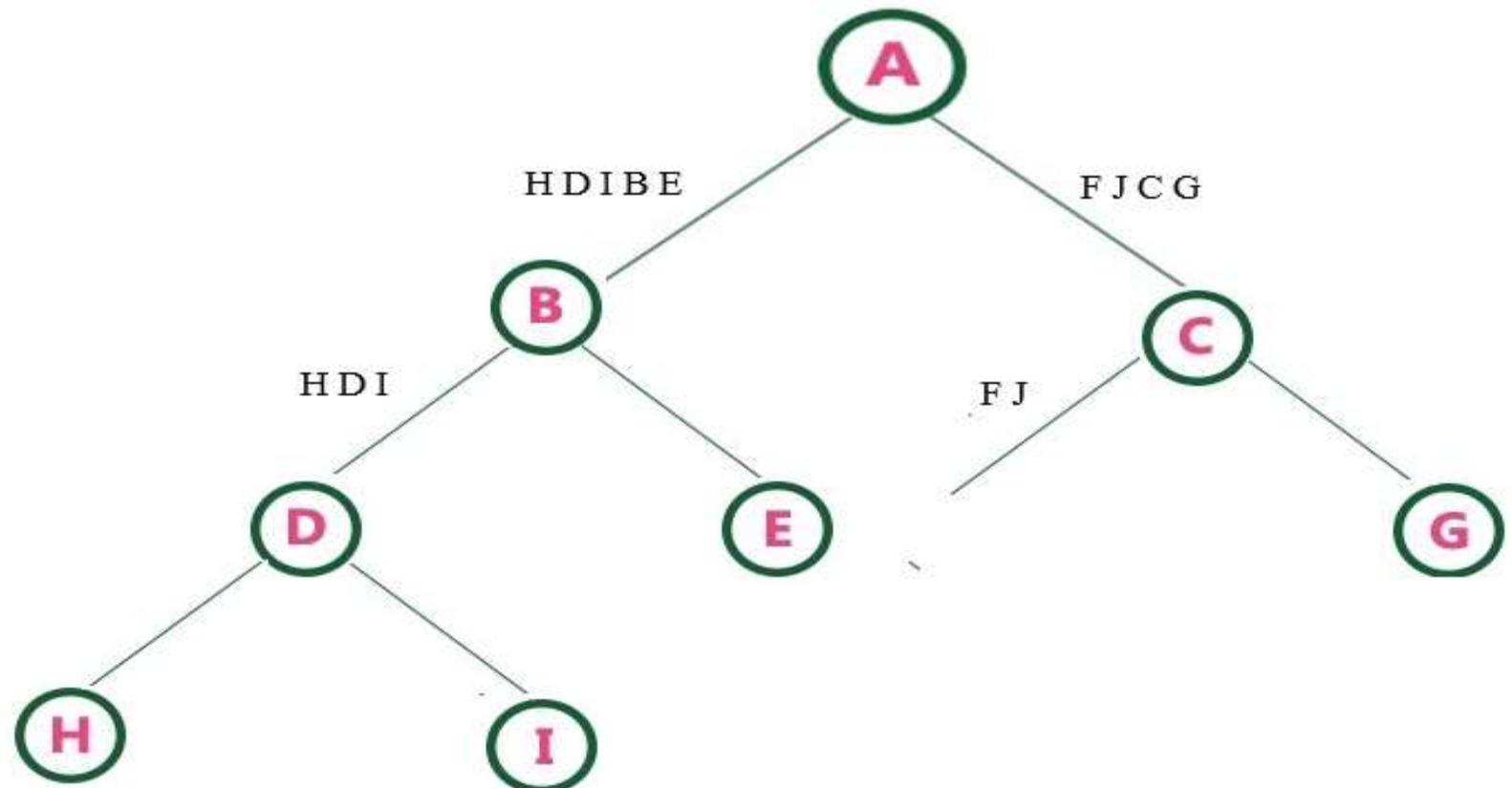
Now LEFT SUB TREE IS
COMPLETED



Now finding the ROOT from POST ORDER
and
LEFT & RIGHT CHILD'S from IN ORDER

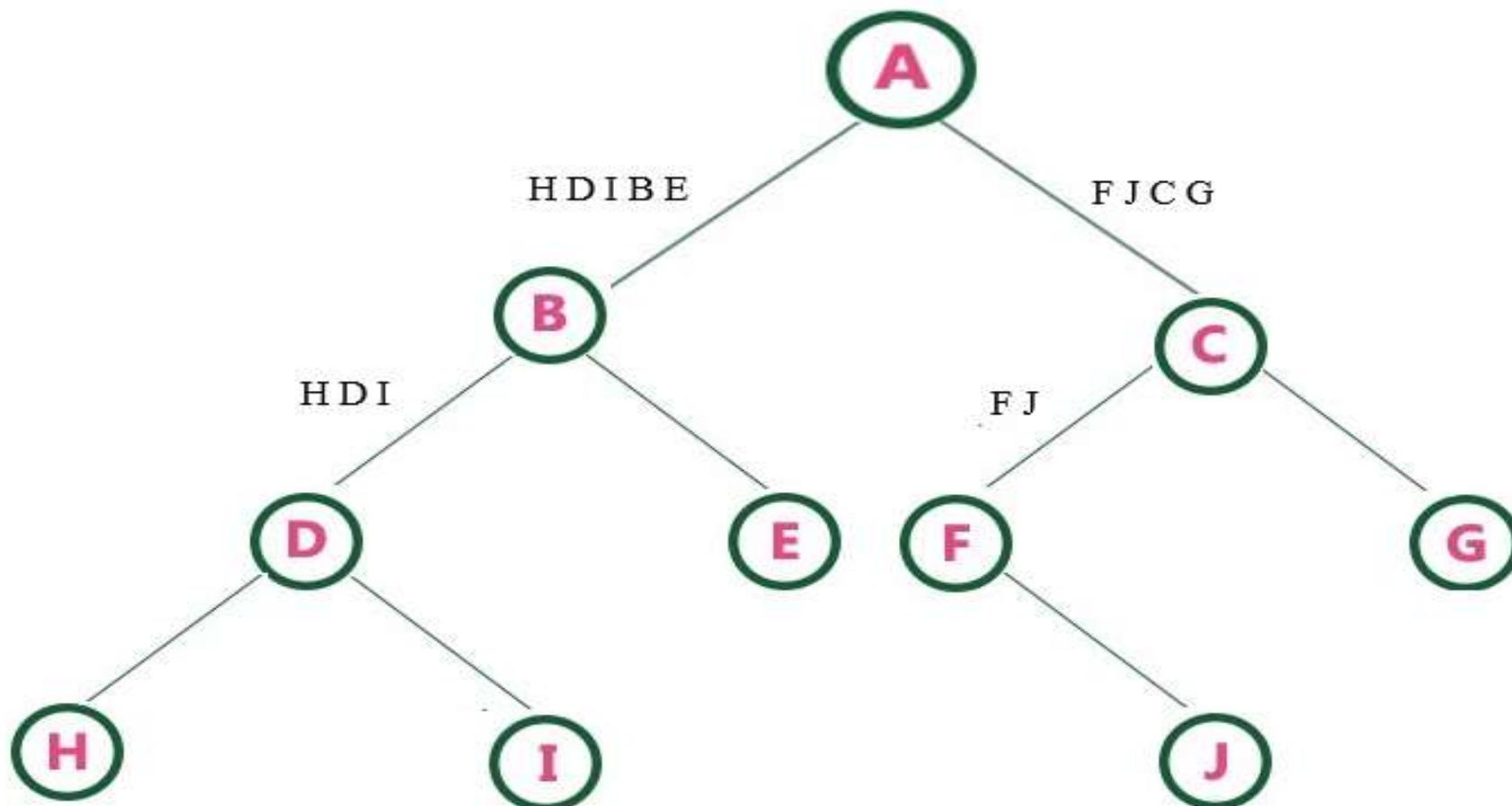
POST Order for the Tree :- H I D E B J F G C A

In Order for the Tree :- H D I B E A F J C G



Now finding the ROOT from POST ORDER
and
LEFT & RIGHT CHILD'S from IN ORDER

POST Order for the Tree :- H I D E B J F G C A
In Order for the Tree :- H D I B E A F J C G



BINARY SEARCH TREES

BINARY TREE + BINARY SEARCH= BINARY SEARCH TREE

BINARY SEARCH TREE PROPERTIES

- A binary search tree (BST), also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in order.
- In a BST, all nodes in the left sub-tree have a value less than that of the root node.
- Correspondingly, all nodes in the right sub-tree have a value either equal to or greater than the root node.
- The same rule is applicable to every sub-tree in the tree.
- Due to its efficiency in searching elements, BSTs are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.

OPERATIONS OF BINARY SEARCH TREE

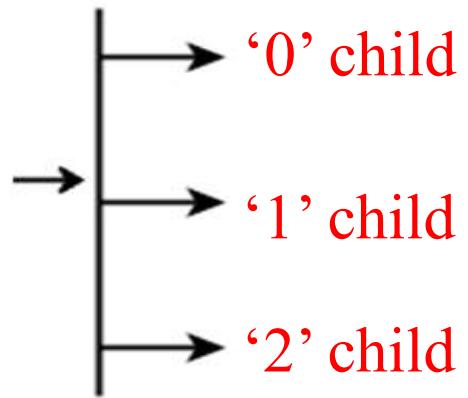
1. INSERTION

2. DELETION

3. SEARCHING

4. FINDING THE MINIMUM ELEMENT

5. FINDING THE MAXIMUM ELEMENT



Searching for a Value in a BST

- The search function is used to find whether a given value is present in the tree or not.
- The function first checks if the BST is empty. If it is, then the value we are searching for is not present in the tree, and the search algorithm terminates by displaying an appropriate message.
- However, if there are nodes in the tree then the search function checks to see if the key value of the current node is equal to the value to be searched.
- If not, it checks if the value to be searched for is less than the value of the node, in which case it should be recursively called on the left child node.
- In case the value is greater than the value of the node, it should be recursively called on the right child node.

Searching Algorithm

```
searchElement (TREE, VAL)
```

Step 1: IF TREE → DATA = VAL OR TREE = NULL

 Return TREE

ELSE

 IF VAL < TREE → DATA

 Return searchElement(TREE → LEFT, VAL)

 ELSE

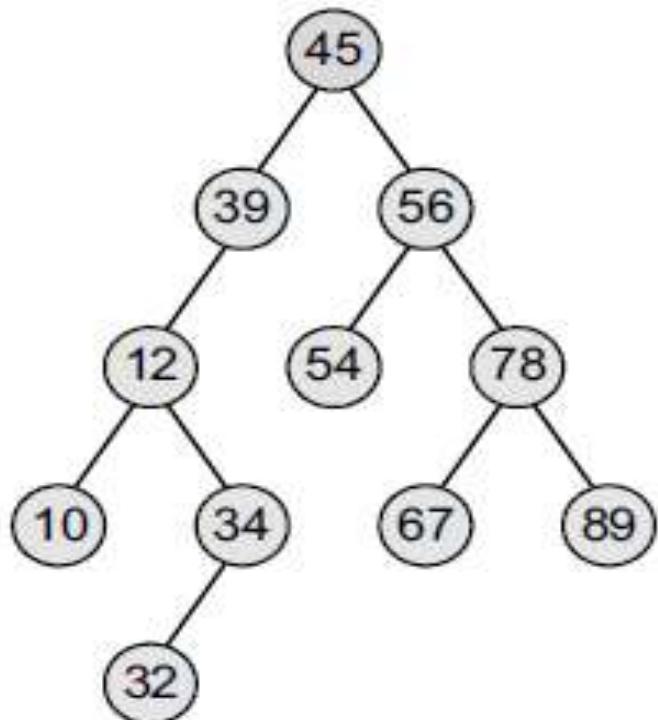
 Return searchElement(TREE → RIGHT, VAL)

 [END OF IF]

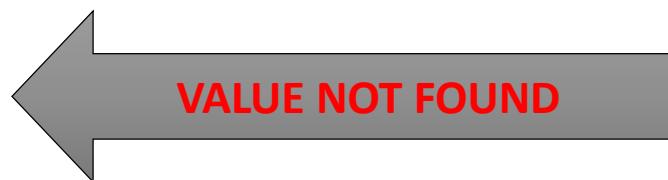
 [END OF IF]

Step 2: END

EXAMPLE



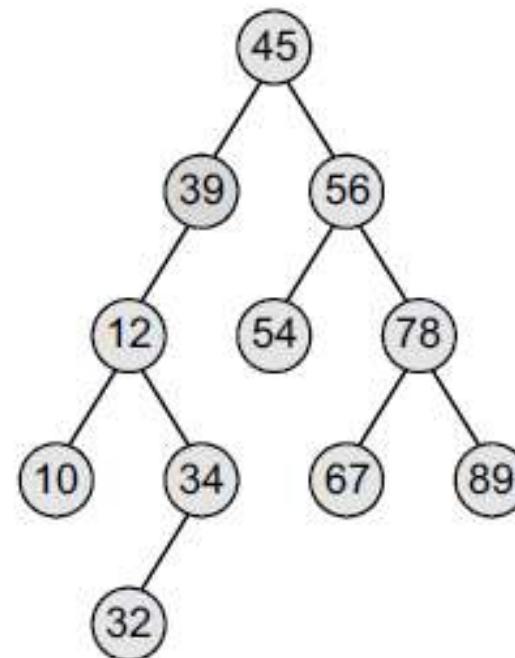
Searching a node with the value 40
in the given binary search tree



Searching a node with the value 67
in the given binary search tree

67

VALUE FOUND



Algorithm to Inserting a node /Value in BST

Insert (TREE, VAL)

Step 1: IF TREE = NULL

 Allocate memory for TREE

 SET TREE → DATA = VAL

 SET TREE → LEFT = TREE → RIGHT = NULL

ELSE

 IF VAL < TREE → DATA

 Insert(TREE → LEFT, VAL)

 ELSE

 Insert(TREE → RIGHT, VAL)

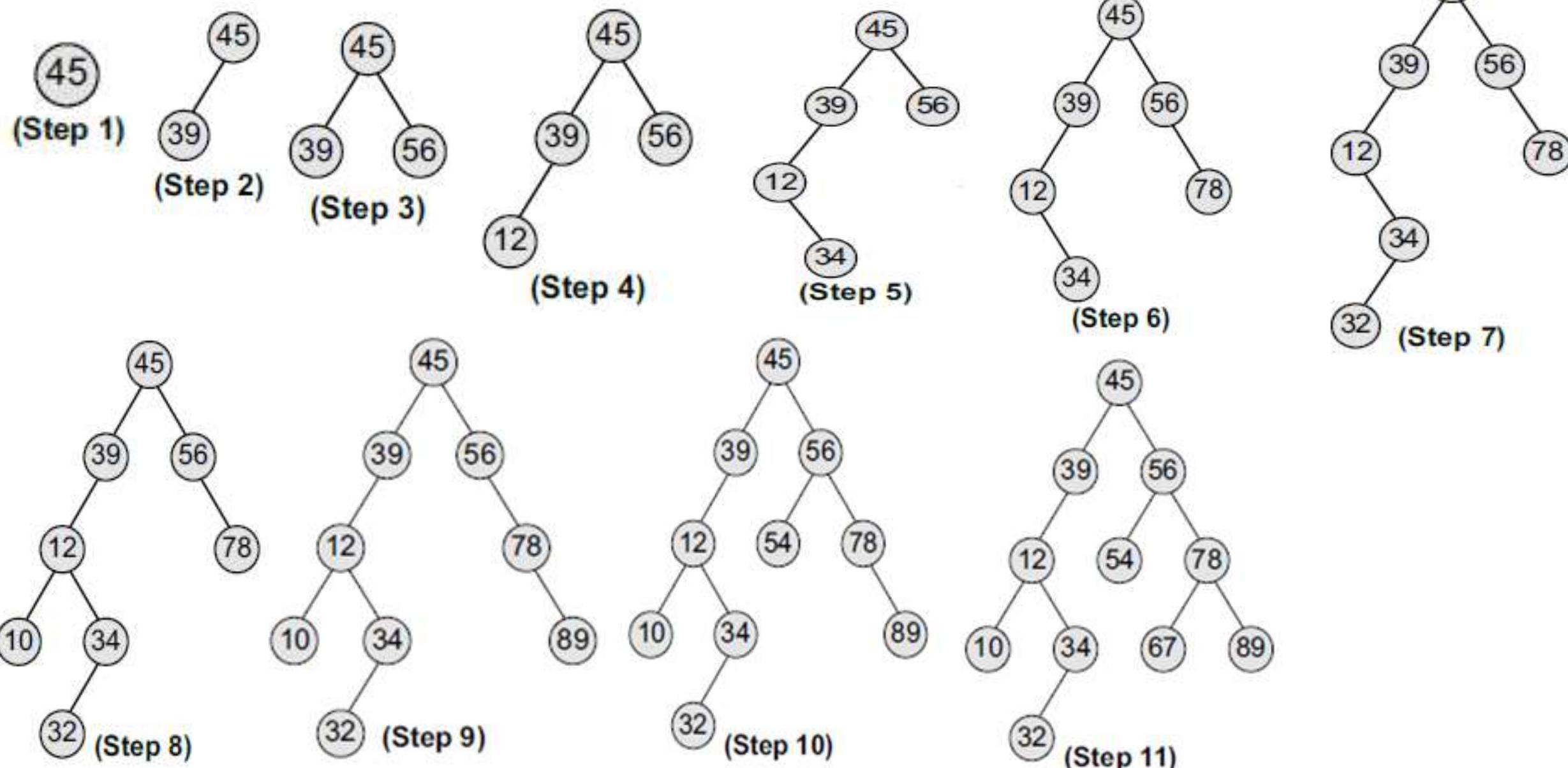
 [END OF IF]

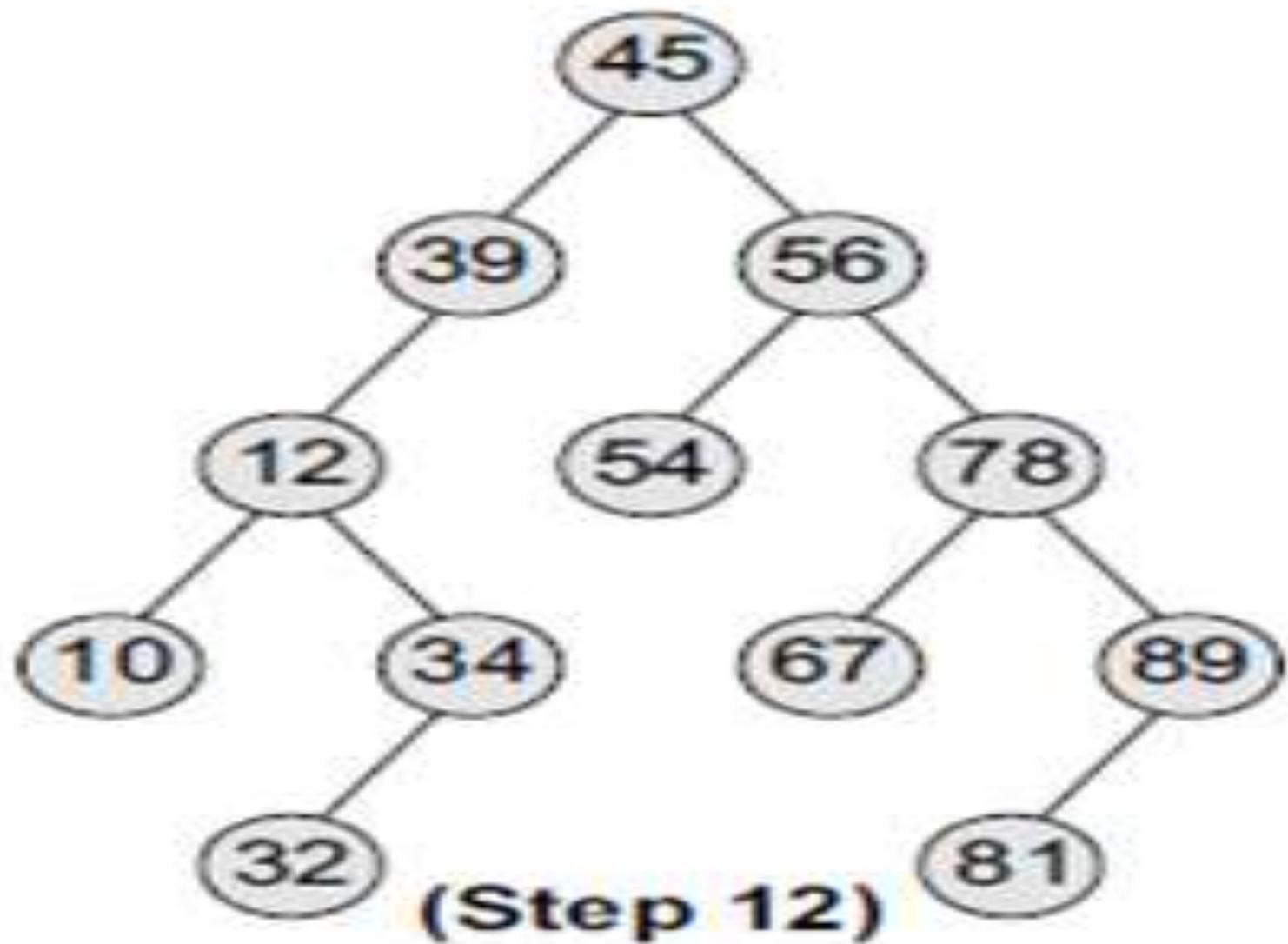
[END OF IF]

Step 2: END

Creating a Binary Search from Given Values

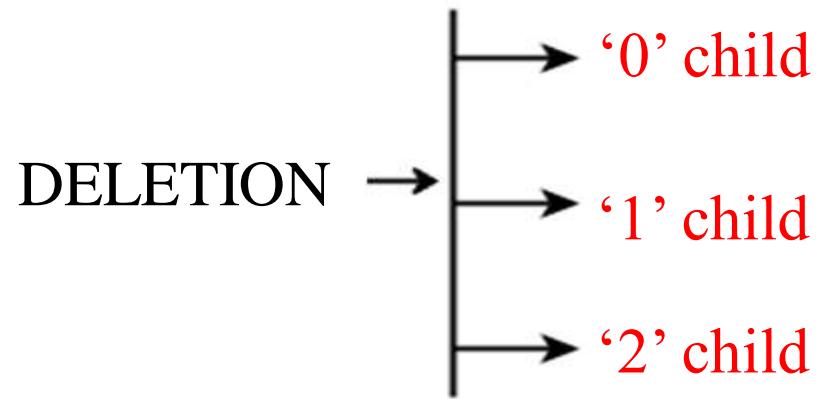
45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67





**DELETING A NODE
IN A
BINARY SEARCH TREE**

DELETNG A NODE IN A BINARY SEARCH TREE



Deleting a node in a binary search tree can be done in **THREE** cases

- Deleting a node with **“no”** children's
- Deleting a node having **“one”** child
- Deleting a node having **“two”** child's

Algorithm for Deleting a node in BST

```
 Delete (TREE, VAL)

Step 1: IF TREE = NULL
        Write "VAL not found in the tree"
        ELSE IF VAL < TREE->DATA
            Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE->DATA
            Delete(TREE->RIGHT, VAL)
        ELSE IF TREE->LEFT AND TREE->RIGHT
            SET TEMP = findLargestNode(TREE->LEFT)
            SET TREE->DATA = TEMP->DATA
            Delete(TREE->LEFT, TEMP->DATA)
        ELSE
            SET TEMP = TREE
            IF TREE->LEFT = NULL AND TREE->RIGHT = NULL
                SET TREE = NULL
            ELSE IF TREE->LEFT != NULL
                SET TREE = TREE->LEFT
            ELSE
                SET TREE = TREE->RIGHT
            [END OF IF]
            FREE TEMP
        [END OF IF]
```

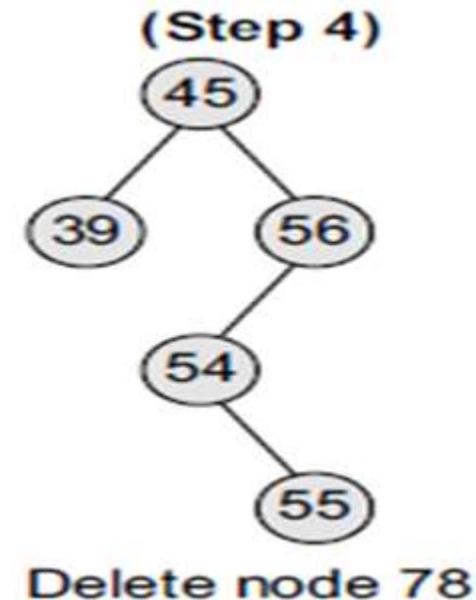
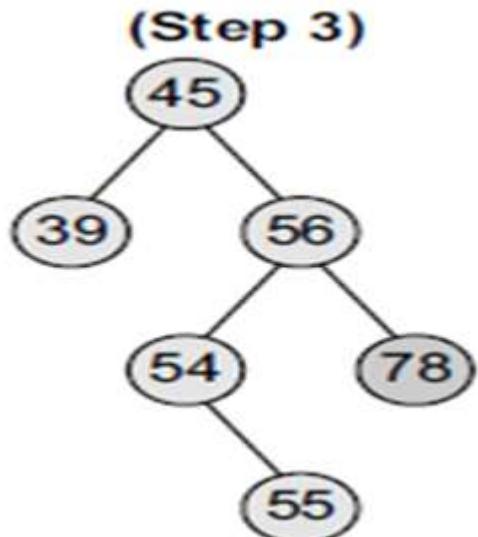
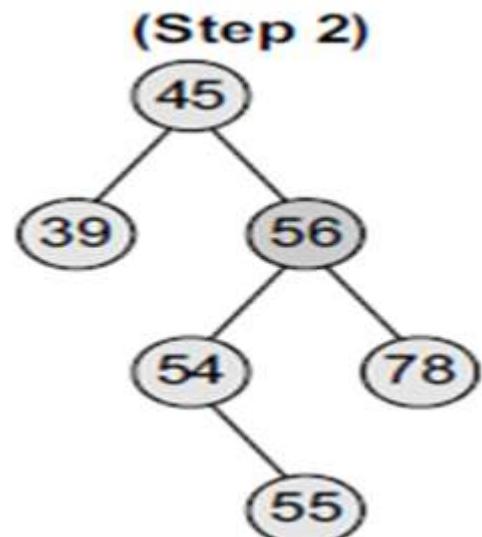
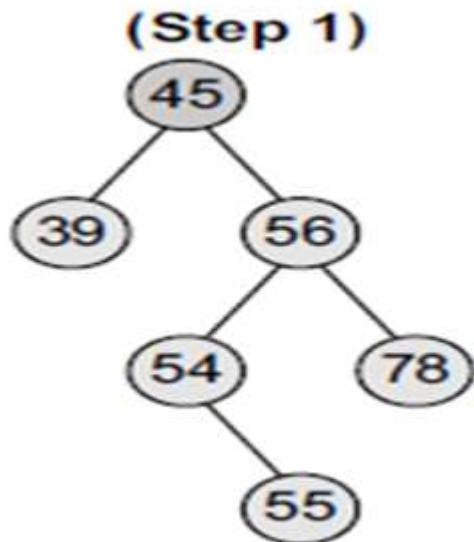
Step 2: END

Deleting a Value from a BST

- The delete function deletes a node from the binary search tree.
- However, care should be taken that the properties of the BSTs do not get violated and nodes are not lost in the process.
- The deletion of a node involves any of the three cases.
- **Case 1:** Deleting a node that has no children.
- **Case 2:** Deleting a node with one child (either left or right).
- To handle the deletion, the node's child is set to be the child of the node's parent.
- Now, if the node was the left child of its parent, the node's child becomes the left child of the node's parent.
- Correspondingly, if the node was the right child of its parent, the node's child becomes the right child of the node's parent.

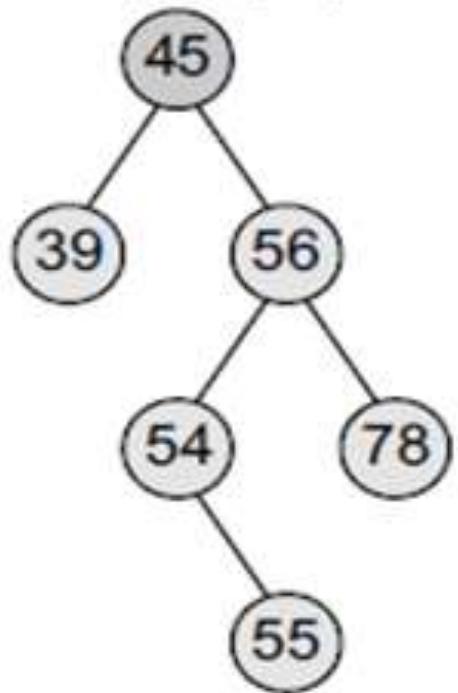
- **Case 3:** Deleting a node with two children.
- To handle this case of deletion, replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree).
- The in-order predecessor or the successor can then be deleted using any of the above cases.

Deleting node 78 from the given binary search tree (No child)

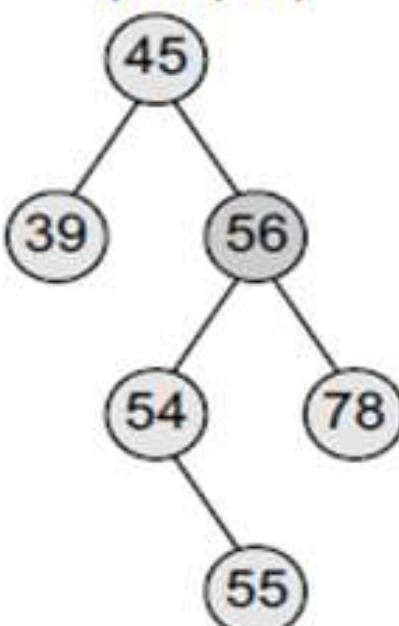


Deleting node 54 from the given binary search tree (one child)

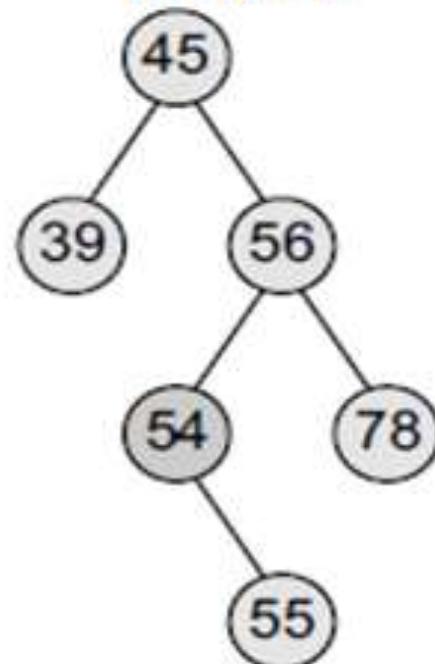
(Step 1)



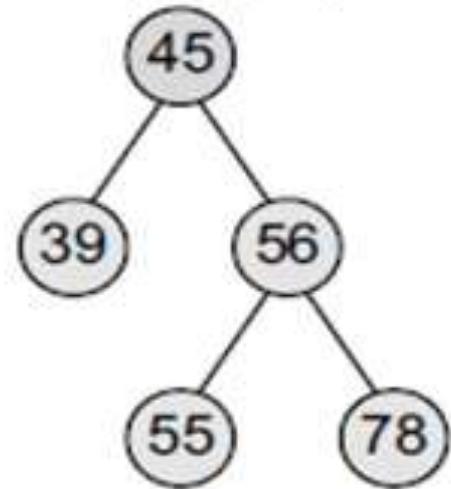
(Step 2)



(Step 3)

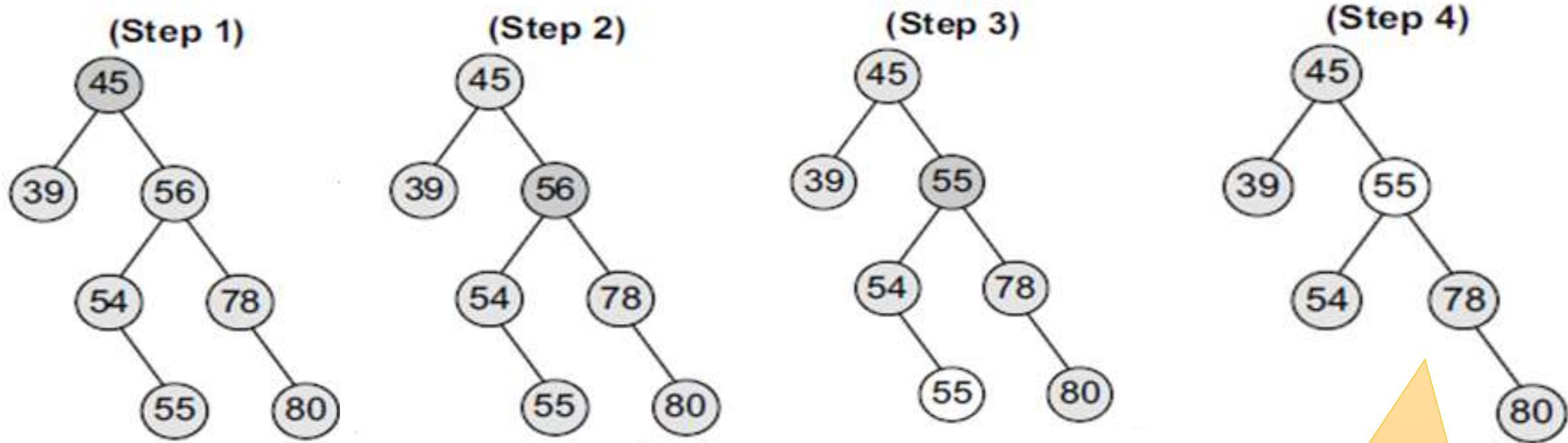


(Step 4)



Replace 54 with 55

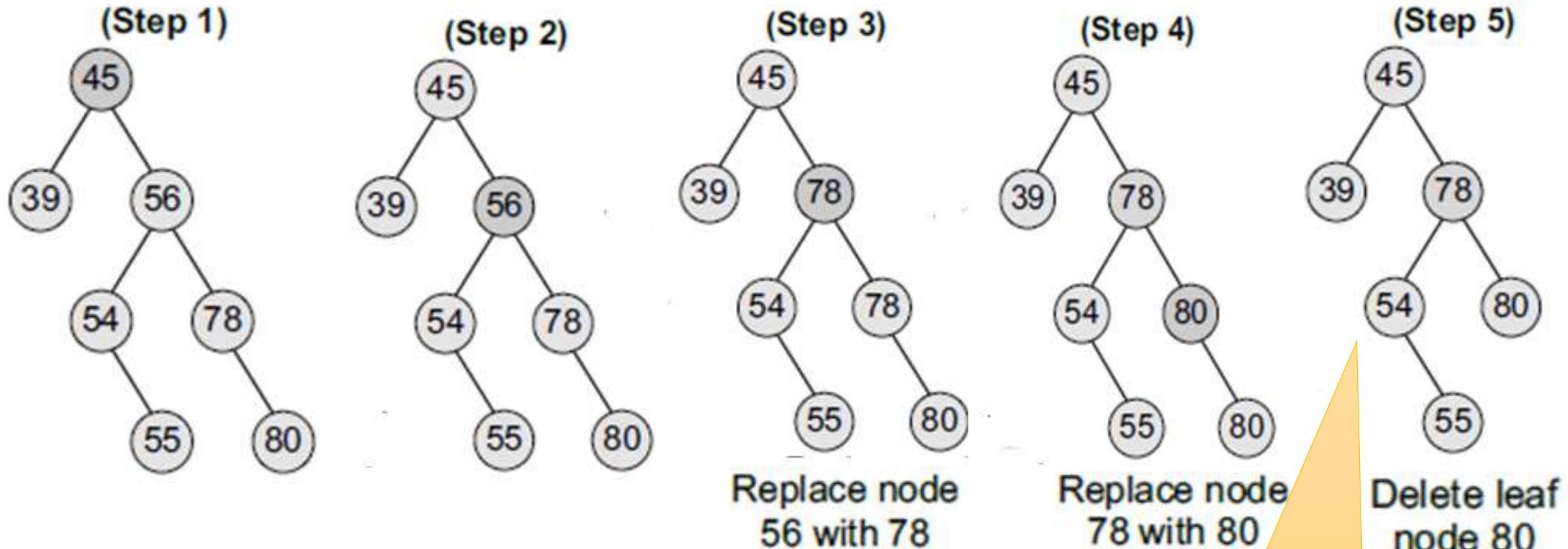
Deleting node 56 from the given binary search tree (Two child)



- Can replace the parent with the child
- Maximum value from the left sub tree

After replacing the
Maximum value
from the left sub tree

Deleting node 56 from the given binary search tree (Two child)



➤ Can replace the parent with the child

➤ Minimum value from the Right Sub Tree

After replacing the
Minimum value from
the Right sub tree

Algorithm to determine the height of a Binary Search Tree

Height (TREE)

Step 1: IF TREE = NULL

 Return 0

ELSE

 SET LeftHeight = Height(TREE → LEFT)

 SET RightHeight = Height(TREE → RIGHT)

 IF LeftHeight > RightHeight

 Return LeftHeight + 1

 ELSE

 Return RightHeight + 1

 [END OF IF]

[END OF IF]

Step 2: END

Algorithm to calculate the number of nodes in a BST

totalNodes(TREE)

Step 1: IF TREE = NULL
 Return 0

 ELSE
 Return totalNodes(TREE → LEFT)
 + totalNodes(TREE → RIGHT) + 1
 [END OF IF]

Step 2: END

Algorithm to calculate the total number of internal nodes in a BST

totalInternalNodes(TREE)

Step 1: IF TREE = NULL
 Return 0
 [END OF IF]
 IF TREE → LEFT = NULL AND TREE → RIGHT = NULL
 Return 0
 ELSE
 Return totalInternalNodes(TREE → LEFT) +
 totalInternalNodes(TREE → RIGHT) + 1
 [END OF IF]

Step 2: END

Algorithm to calculate the total number of External nodes in a BST

totalExternalNodes(TREE)

Step 1: IF TREE = NULL

 Return 0

 ELSE IF TREE → LEFT = NULL AND TREE → RIGHT = NULL

 Return 1

 ELSE

 Return totalExternalNodes(TREE → LEFT) +

 totalExternalNodes(TREE → RIGHT)

 [END OF IF]

Step 2: END

Algorithm To Delete A Binary Search Tree

```
deleteTree(TREE)

Step 1: IF TREE != NULL
        deleteTree (TREE -> LEFT)
        deleteTree (TREE -> RIGHT)
        Free (TREE)
    [END OF IF]
Step 2: END
```

Algorithm to find the smallest element in a BST

```
findSmallestElement(TREE)

Step 1: IF TREE = NULL OR TREE->LEFT = NULL
        Return TREE
    ELSE
        Return findSmallestElement(TREE->LEFT)
    [END OF IF]
Step 2: END
```

Algorithm to find the Largest element in a BST

findLargestElement(TREE)

Step 1: IF TREE = NULL OR TREE → RIGHT = NULL
 Return TREE

 ELSE

 Return findLargestElement(TREE → RIGHT)

 [END OF IF]

Step 2: END

COUNT BINARY TREE

The recursive structure of a binary tree makes it easy to count nodes recursively. There are 3 things we can count:

- ✓ **The total number of nodes**
- ✓ **The number of leaf nodes**
- ✓ **The number of internal nodes**

Counting All Nodes

The number of nodes in a binary tree is the number of nodes in the root's left sub tree, plus the number of nodes in its right sub tree, plus one (for the root itself).

Counting Leaf Nodes

This is similar, except that we only return 1 if we are a leaf node. Otherwise, we recursive into the left and right sub trees and return the sum of the leaves in them.

Counting Internal Nodes

This is the counterpart of counting leaves. If we are an internal node, we count 1 for ourselves, then recursive into the left and right sub trees and sum the count of internal nodes in them.

Counting Internal Nodes

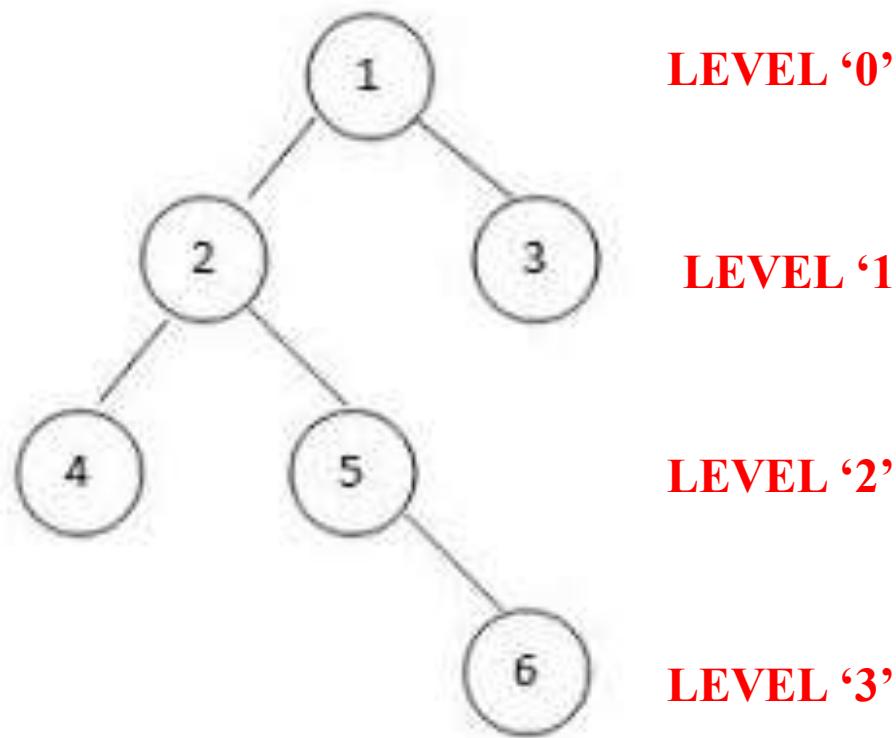
TWO WAYS

1. Iterative Way

2. Recursive Way

- ✓ IT is a Level Order Traversal
- ✓ First initialize the COUNT variable with value = 0
- ✓ Then COUNT =COUNT + 1

Iterative Way



LEVEL 0

1					
0	1	2	3	4	5

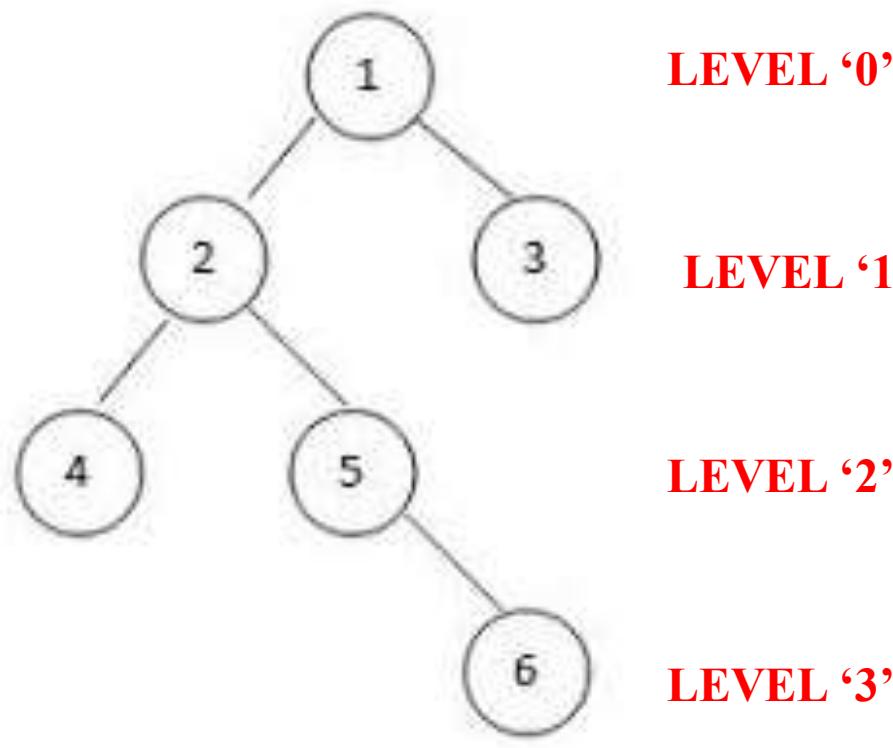
COUNT = 0

COUNT = COUNT+1

COUNT = 0+1

= 1

Iterative Way



LEVEL '0'

LEVEL '1'

LEVEL '2'

LEVEL '3'

LEVEL 1

COUNT = COUNT+1

1	2	3			
0	1	2	3	4	5

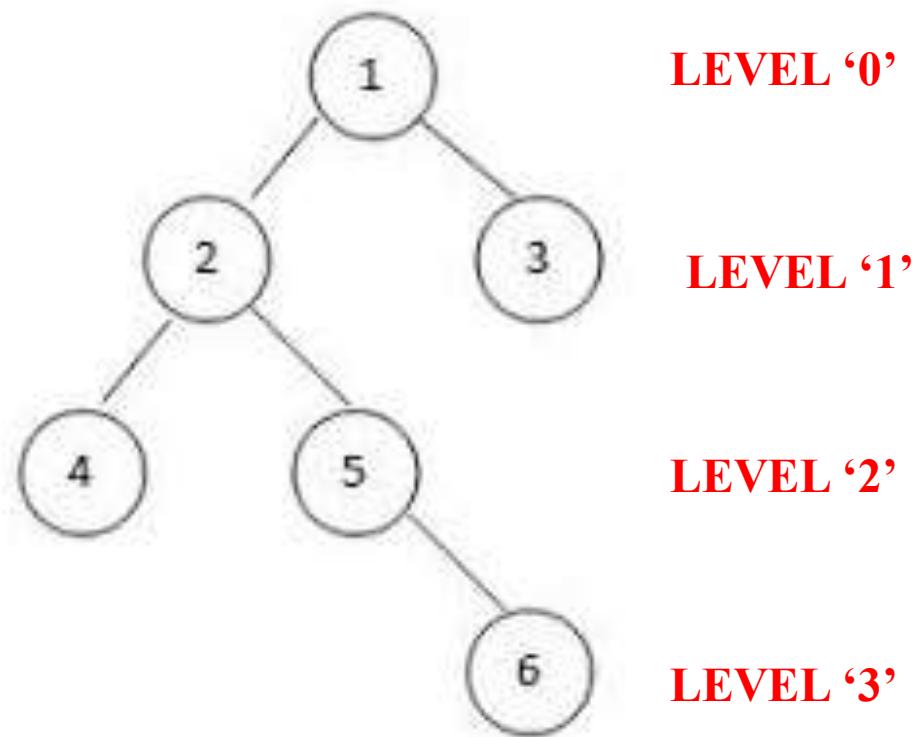
= 2

COUNT = 1+1

COUNT = 2+1

= 3

Iterative Way



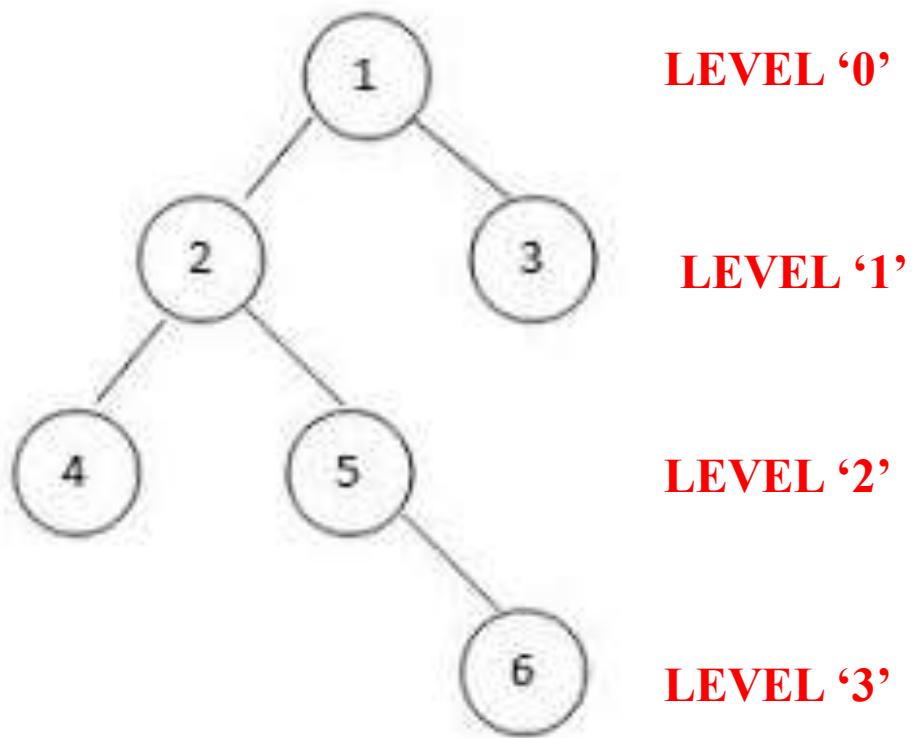
COUNT = COUNT+1

1	2	3	4	5	
0	1	2	3	4	5

$$= 4$$

$$\begin{aligned} \text{COUNT} &= 4+1 \\ &= 5 \end{aligned}$$

Iterative Way



COUNT = COUNT+1

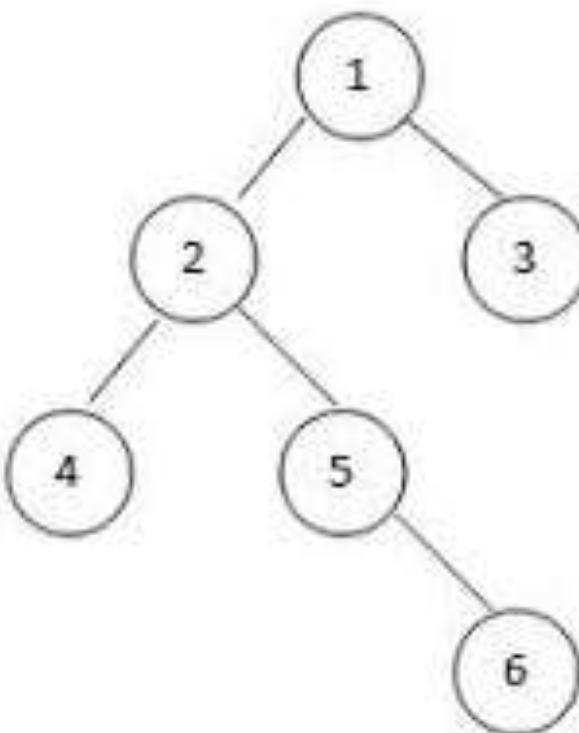
1	2	3	4	5	6
0	1	2	3	4	5

COUNT = 5+1
= 6

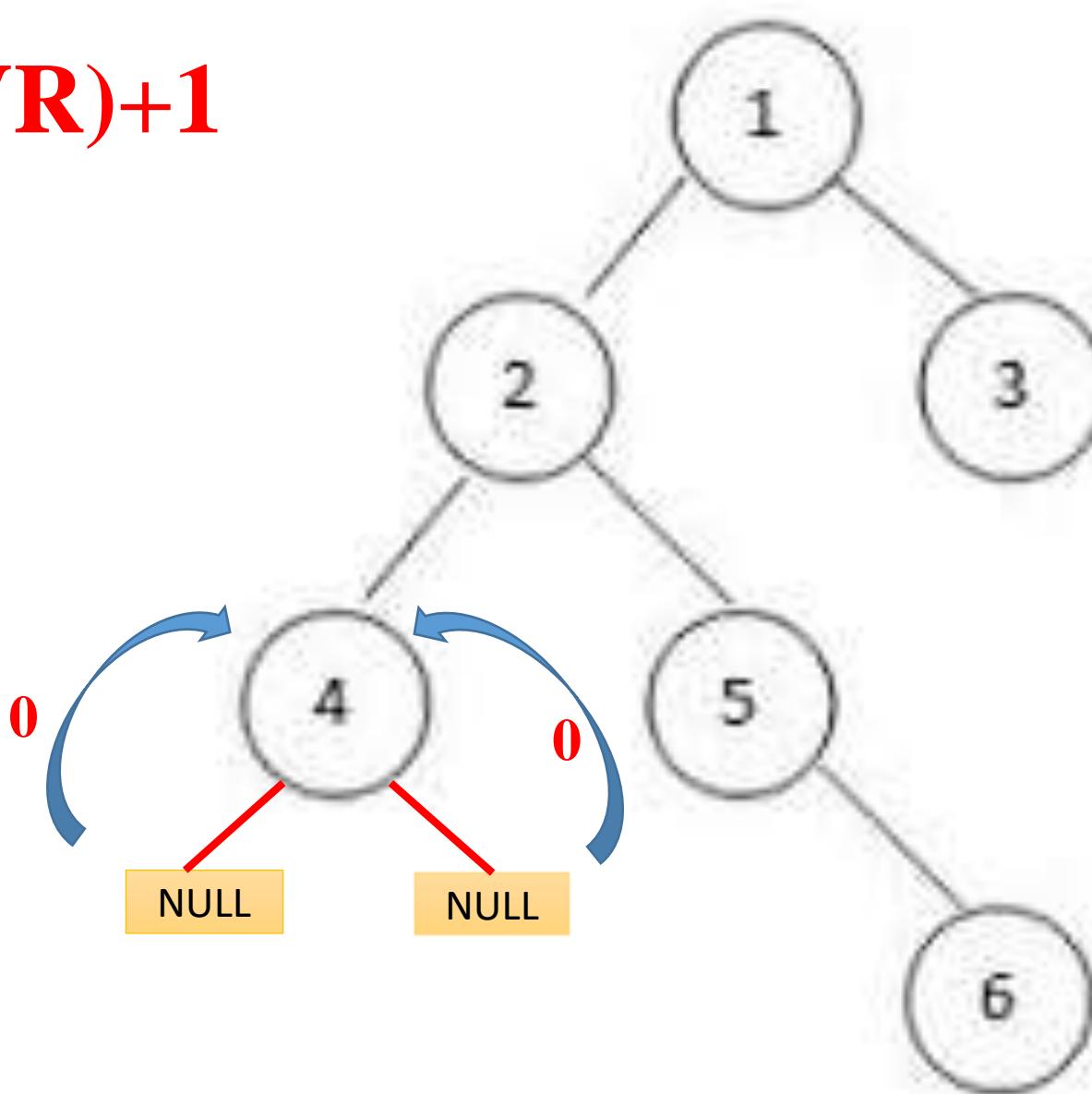
Recursive Way

To find the Recursive Way we use the formula $(VL+VR)+1$

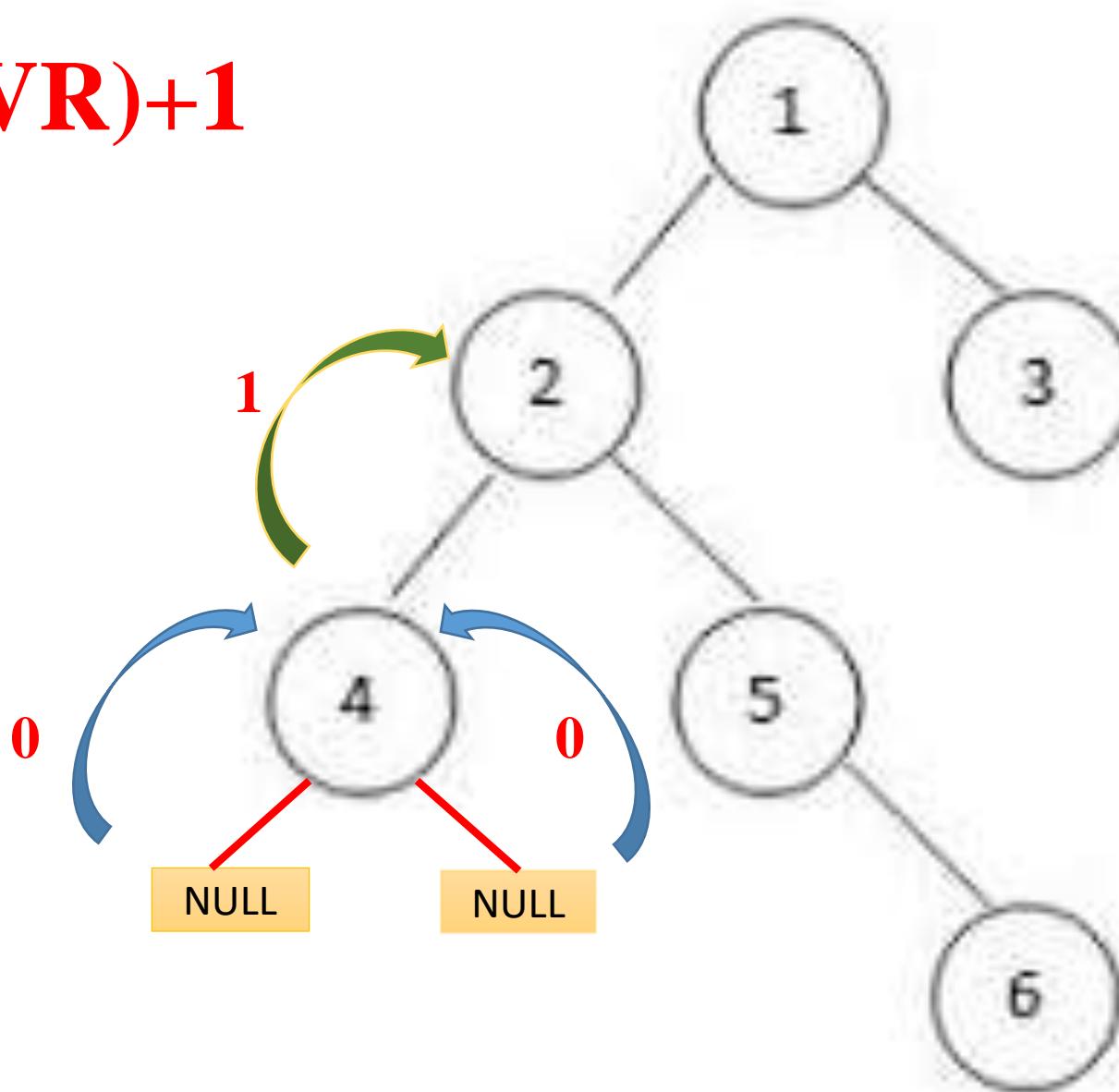
When we will hit the **NULL** then we will apply the formula



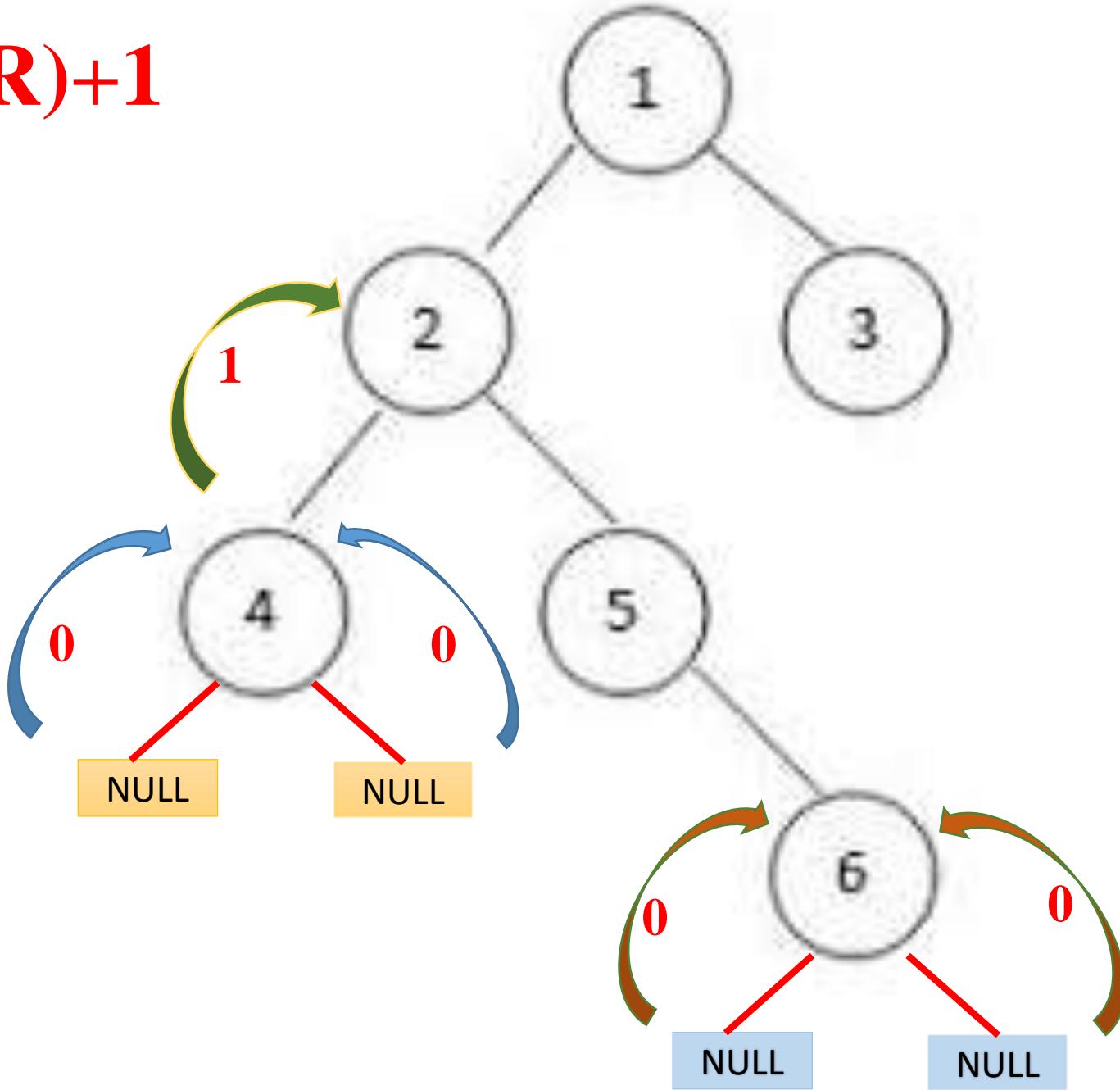
$(VL+VR)+1$



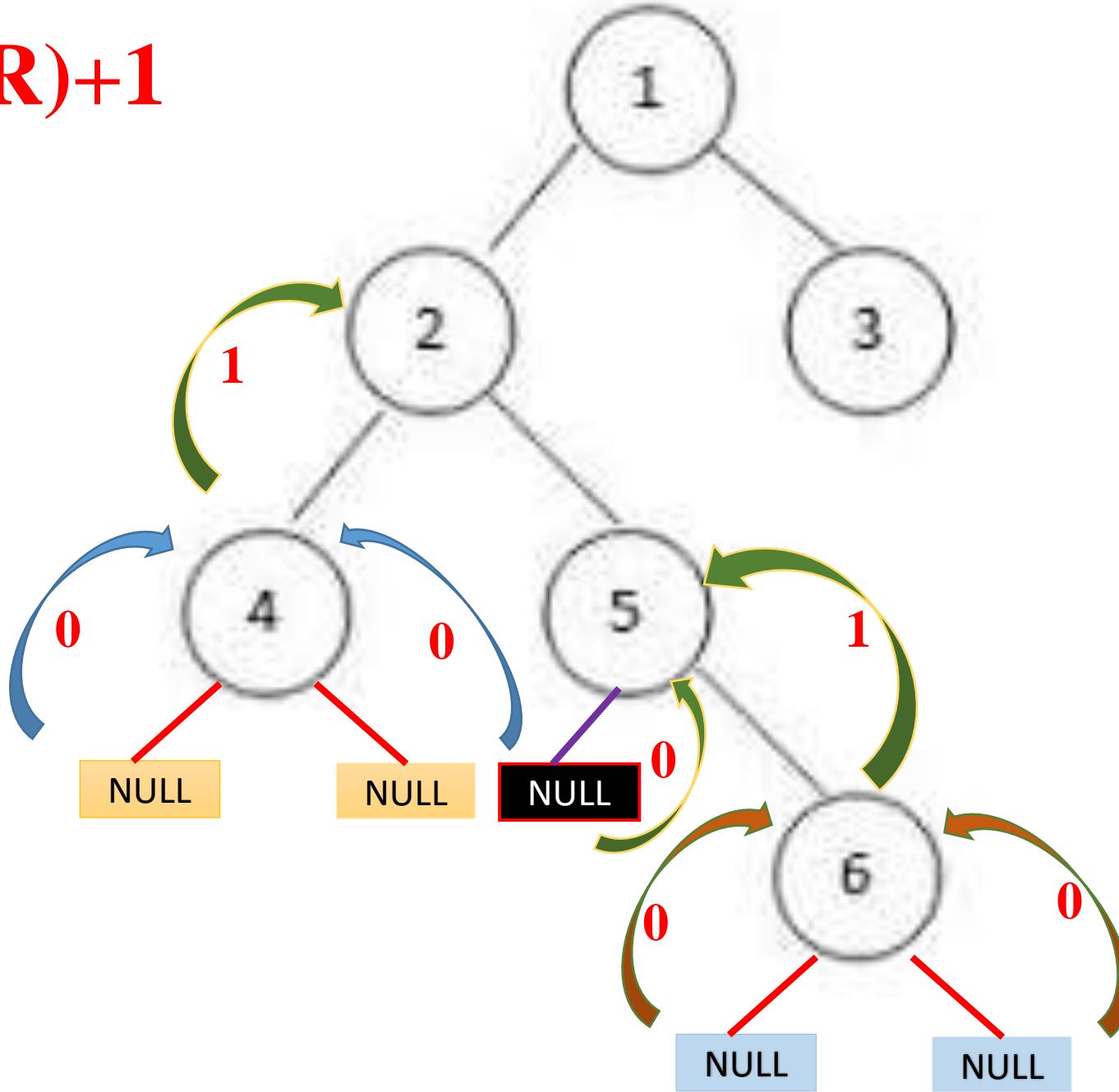
$(VL+VR)+1$



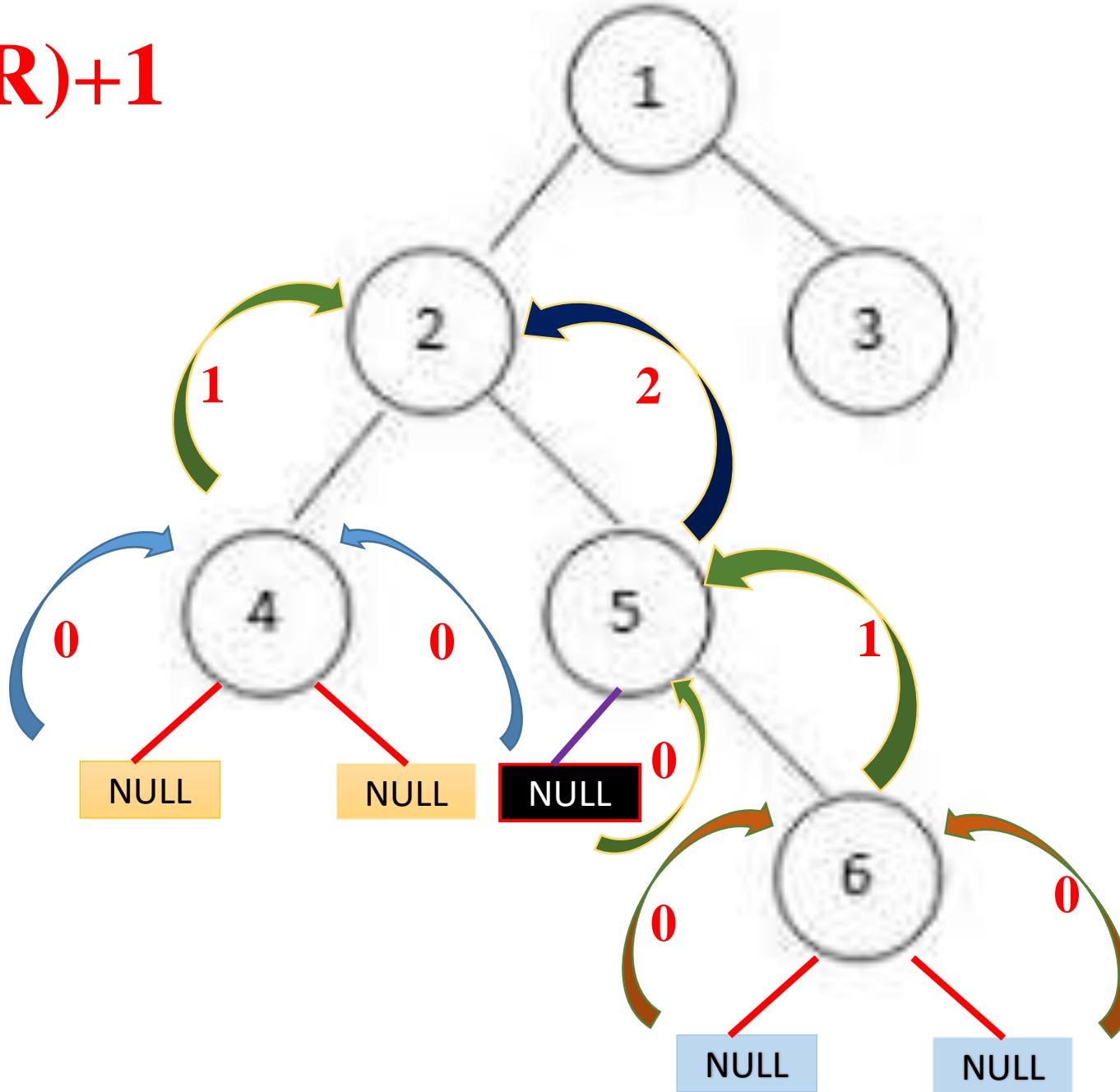
$$(VL+VR)+1$$



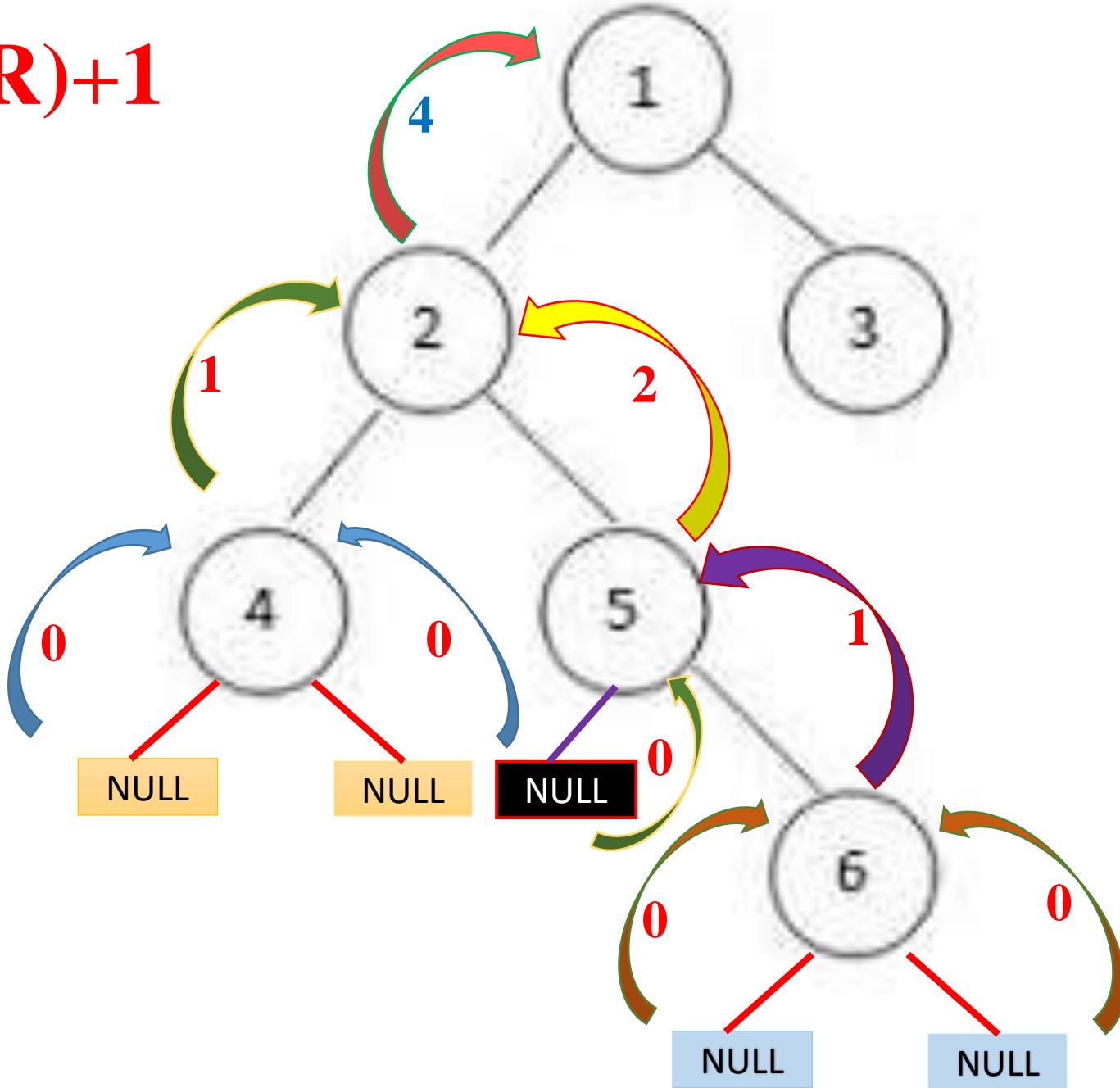
$$(VL+VR)+1$$



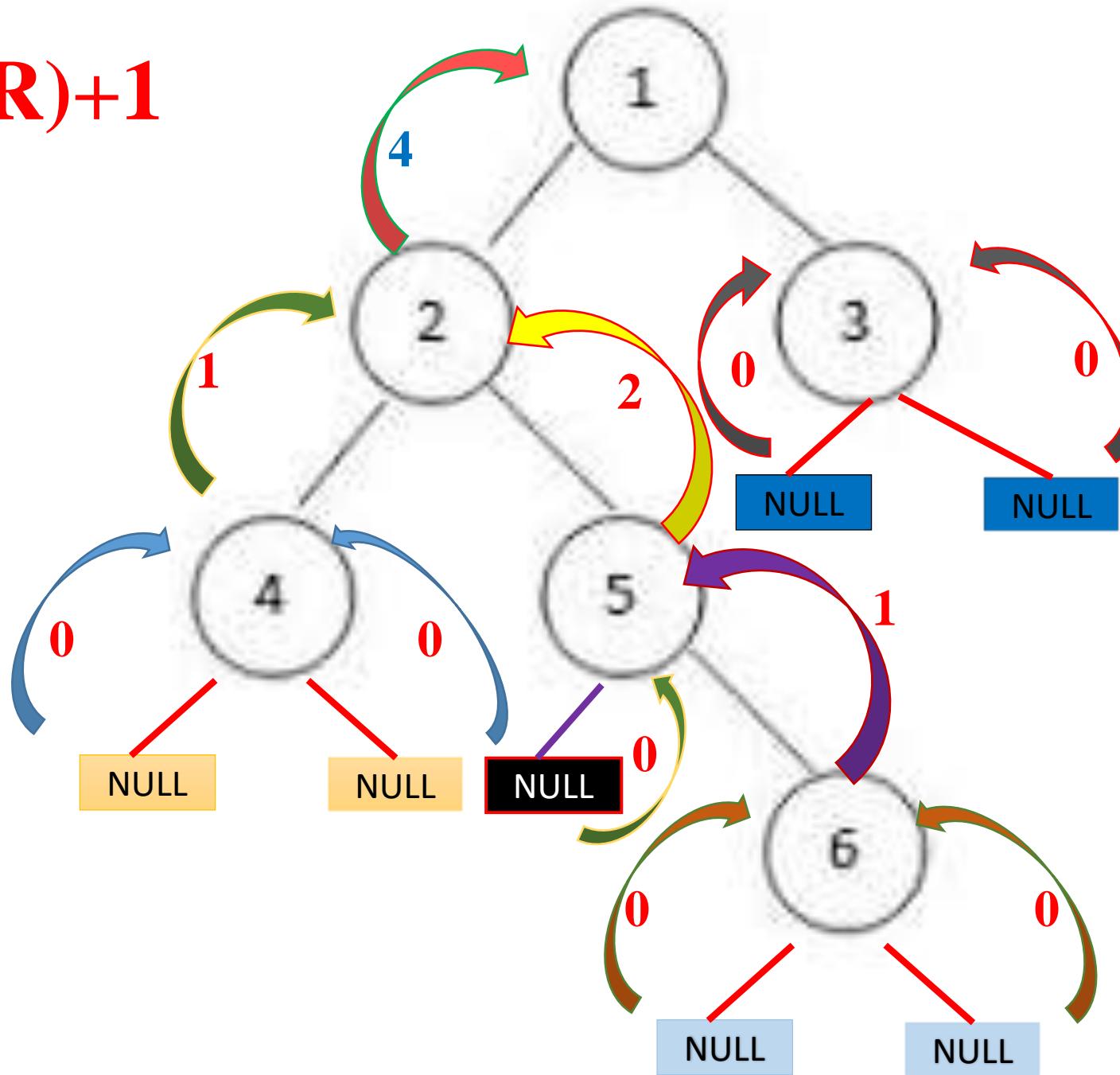
$(VL+VR)+1$



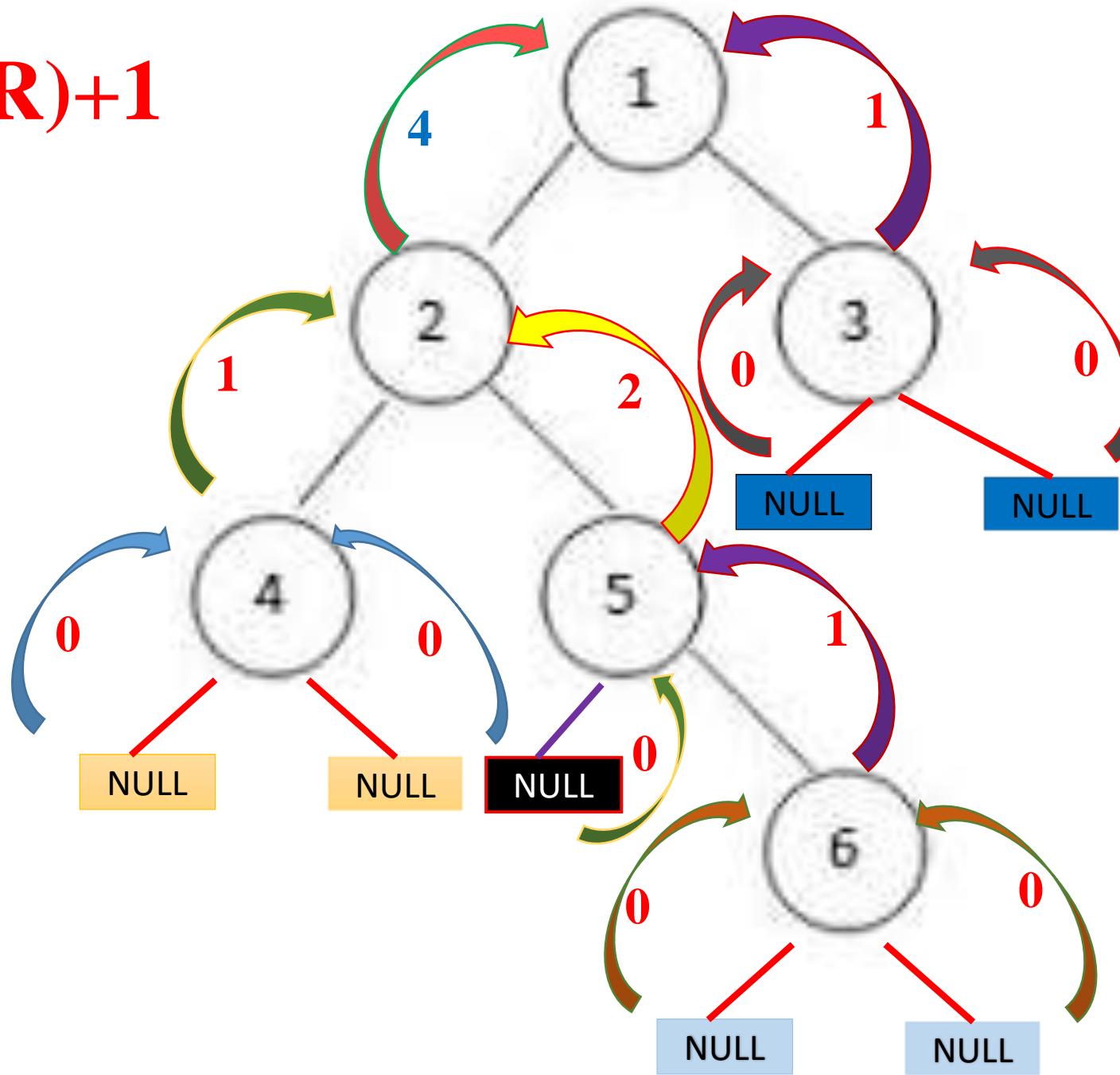
$(VL+VR)+1$



$(VL+VR)+1$



$(VL+VR)+1$



EXPRESSION TREES

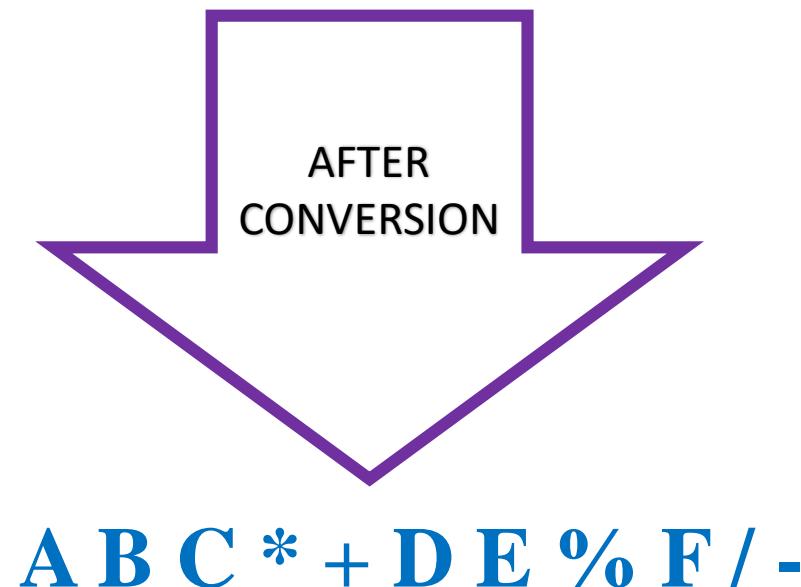
ALGORITHM

- ✓ Every operator must be an **INTERNAL** node (not a leaf node)
- ✓ Every operand must be in **EXTERNAL** node (a leaf node)
- ✓ Convert the **INFIX** expression into **PRE-FIX** or **POST- FIX** expression.
- ✓ Read all the characters from **PRE-FIX** or **POST- FIX** expression.
- ✓ If the character is a “**OPERAND**” than push it into a **STACK**.
- ✓ If the character is a “**OPERATOR**” than.
 - POP the top most two elements from the stack and perform the required operation.
 - Consider the operator is a root node and construct a tree with left and right child's.
 - After constructing a tree then, **PUSH** the tree as element into a stack.
- ✓ Repeat the process until reading all characters from **PRE-FIX** or **POST- FIX** expression.

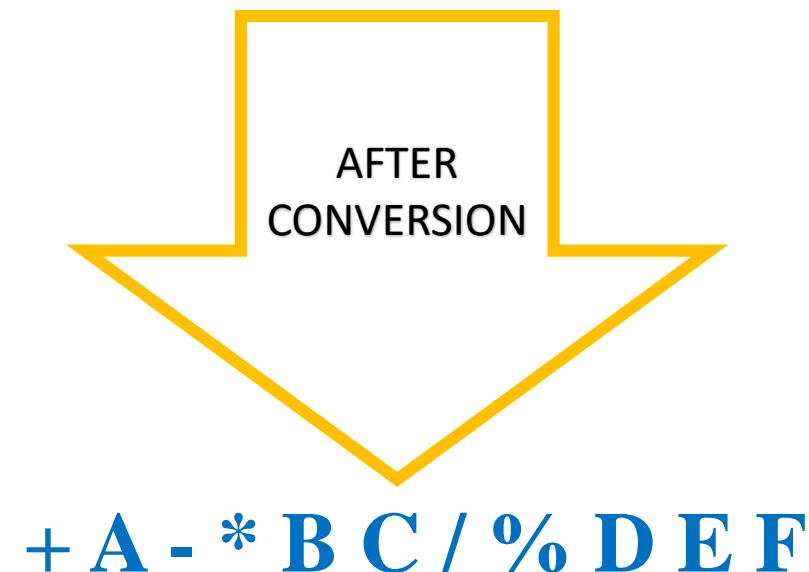
$$A + B * C - D \% E / F$$

Now Convert the expression into **PREFIX** or **POSTFIX**

POST-FIX Expression

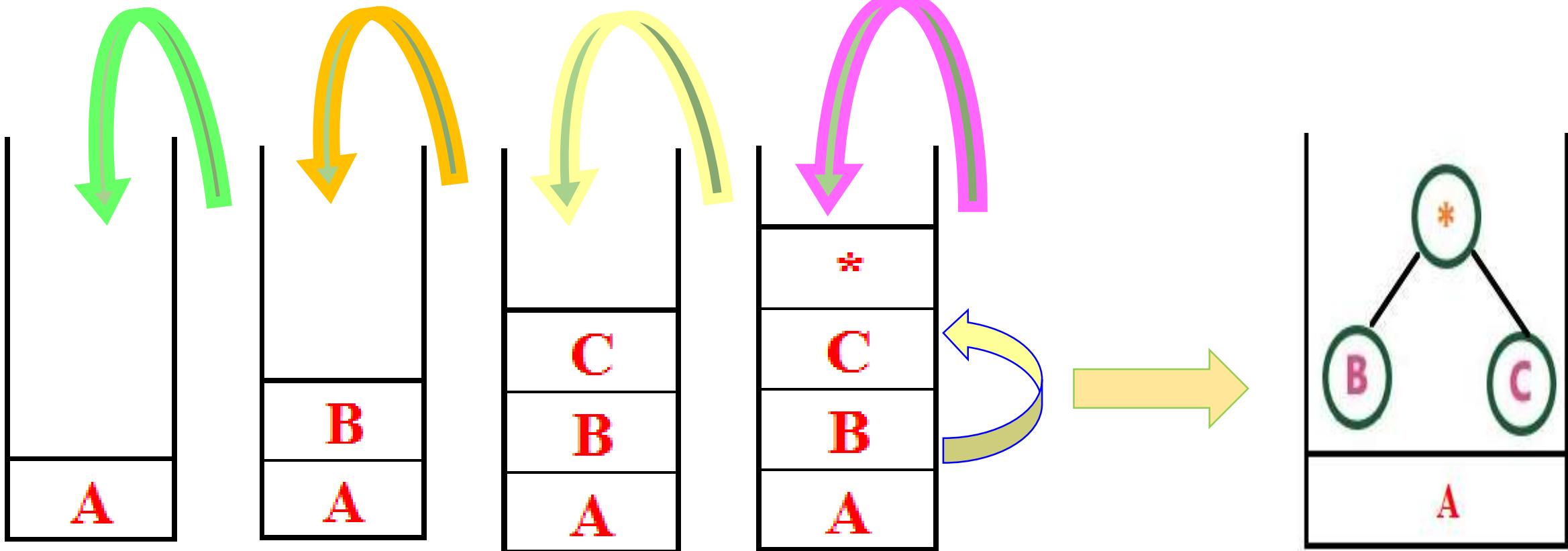


PRE-FIX Expression



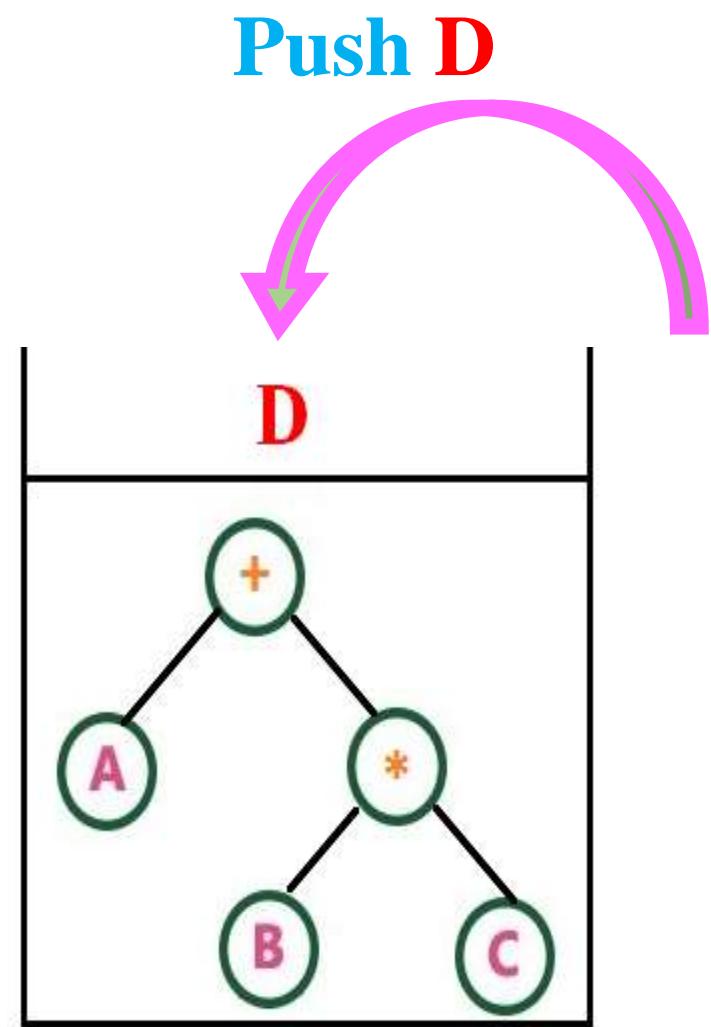
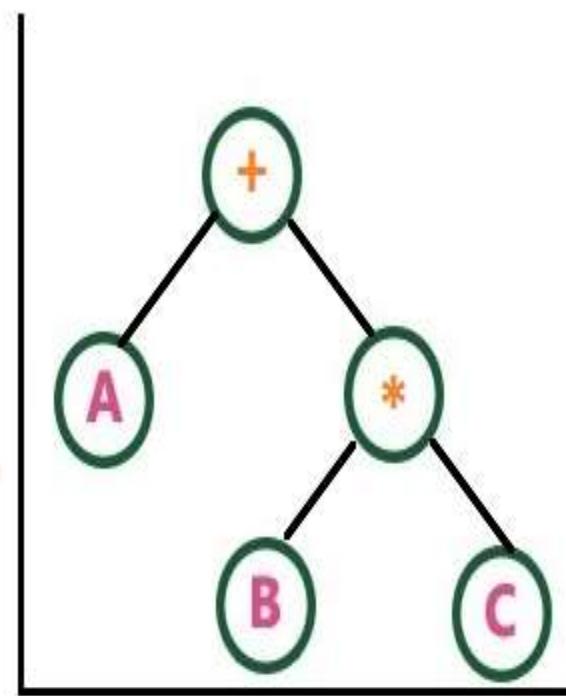
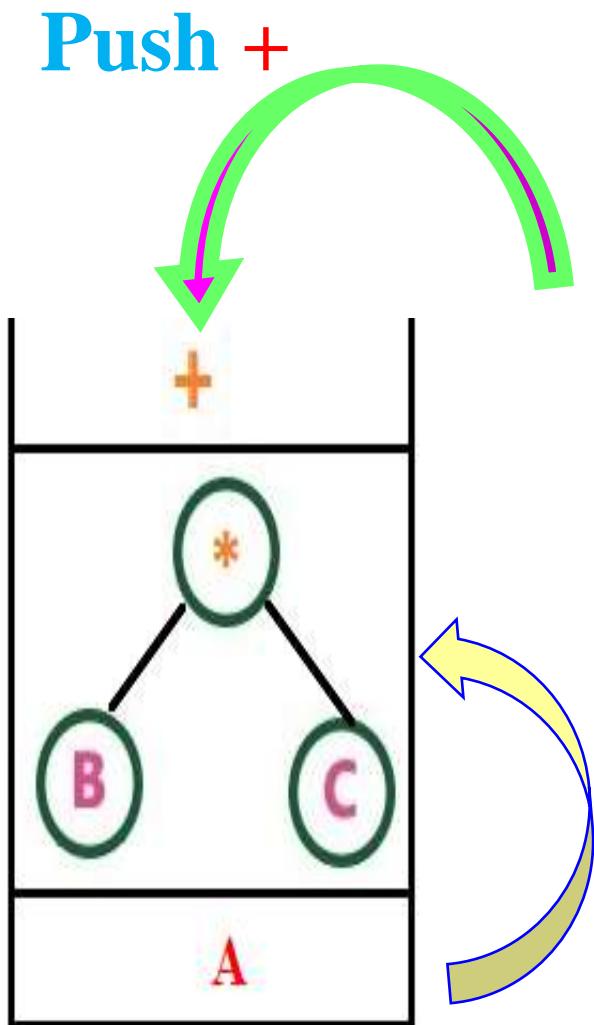
FROM THE POST FIX EXPRESSION **A B C * + D E % F / -**
↑↑↑↑↑

Push A Push B Push C Push *

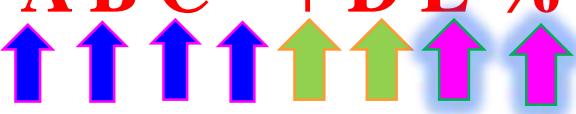


FROM THE POST FIX EXPRESSION **A B C * + D E % F / -**

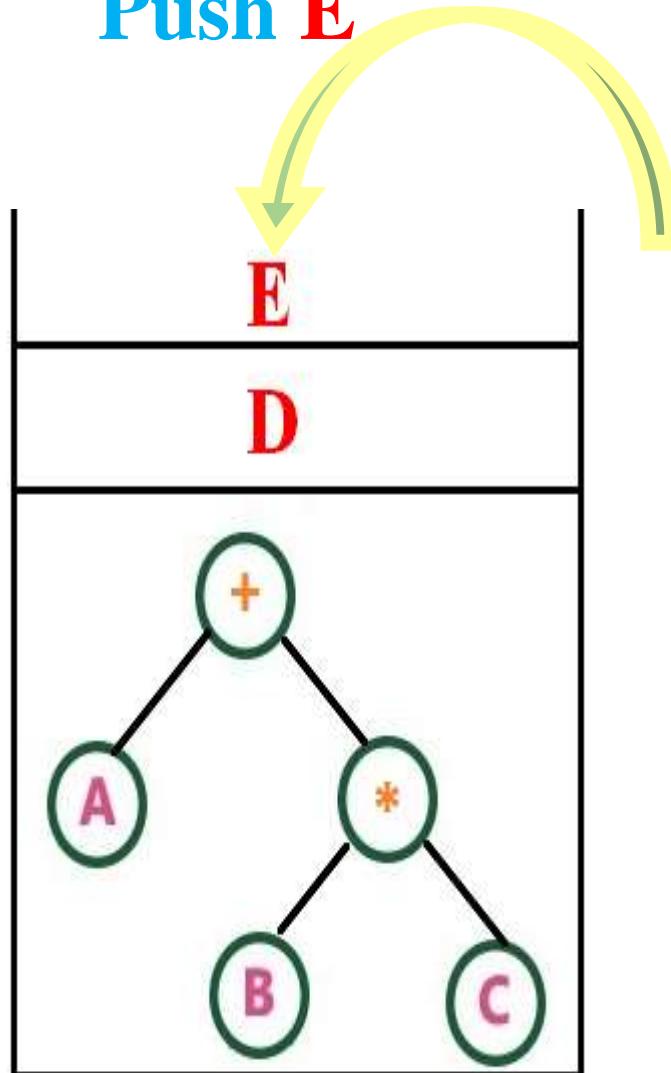
↑↑↑↑↑↑↑



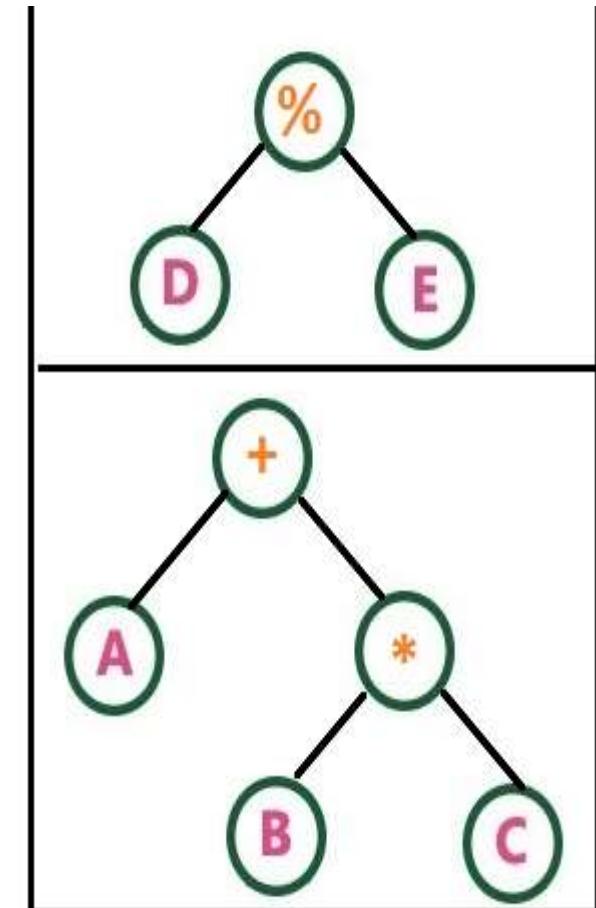
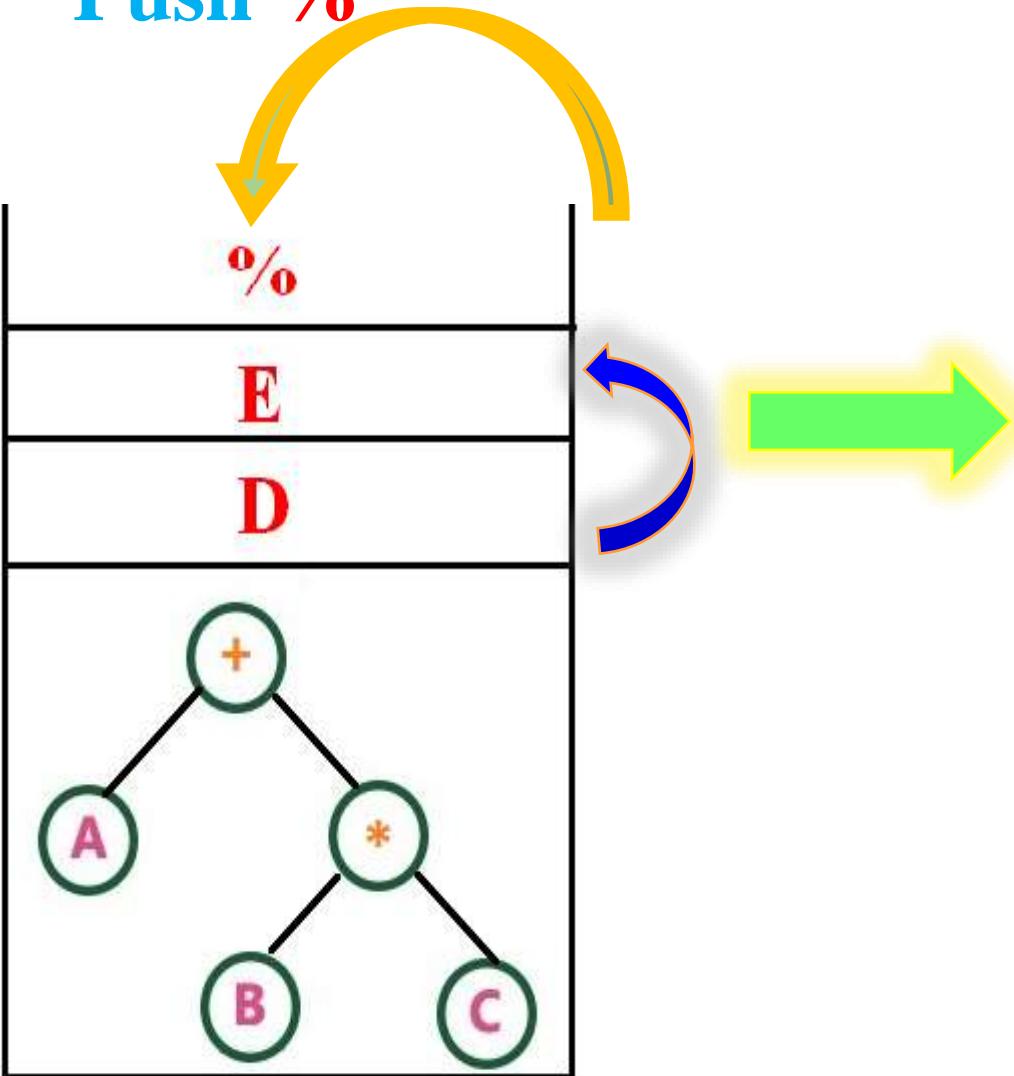
FROM THE POST FIX EXPRESSION **A B C * + D E % F / -**



Push E



Push %

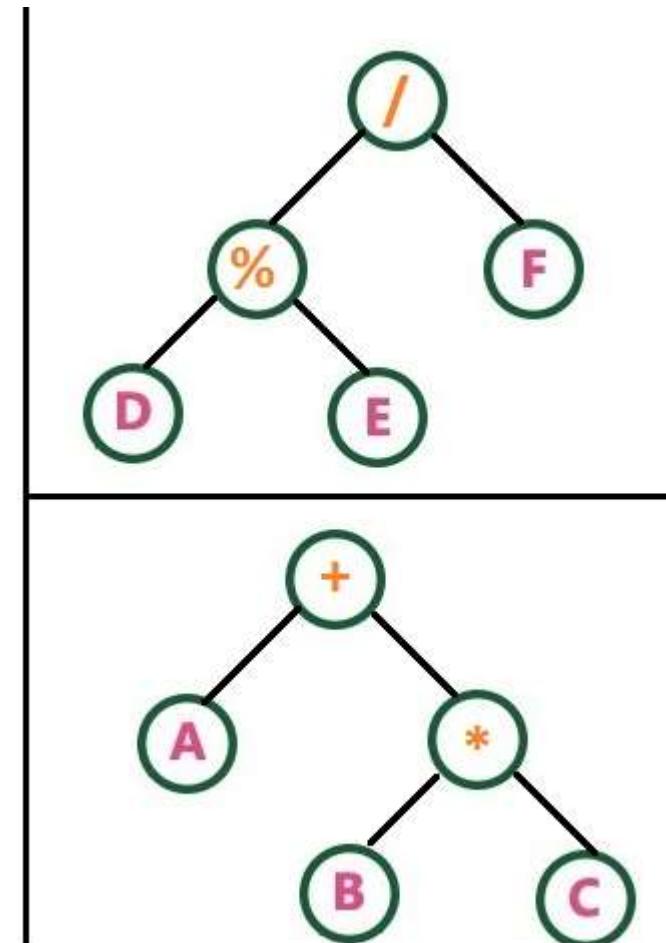
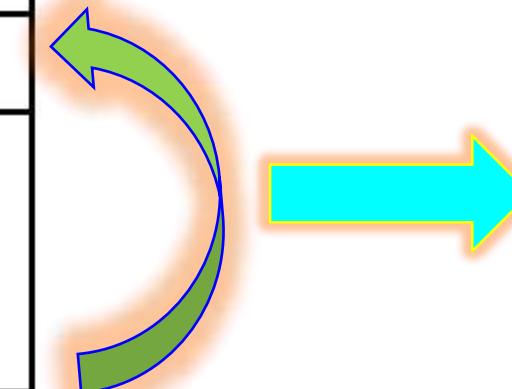
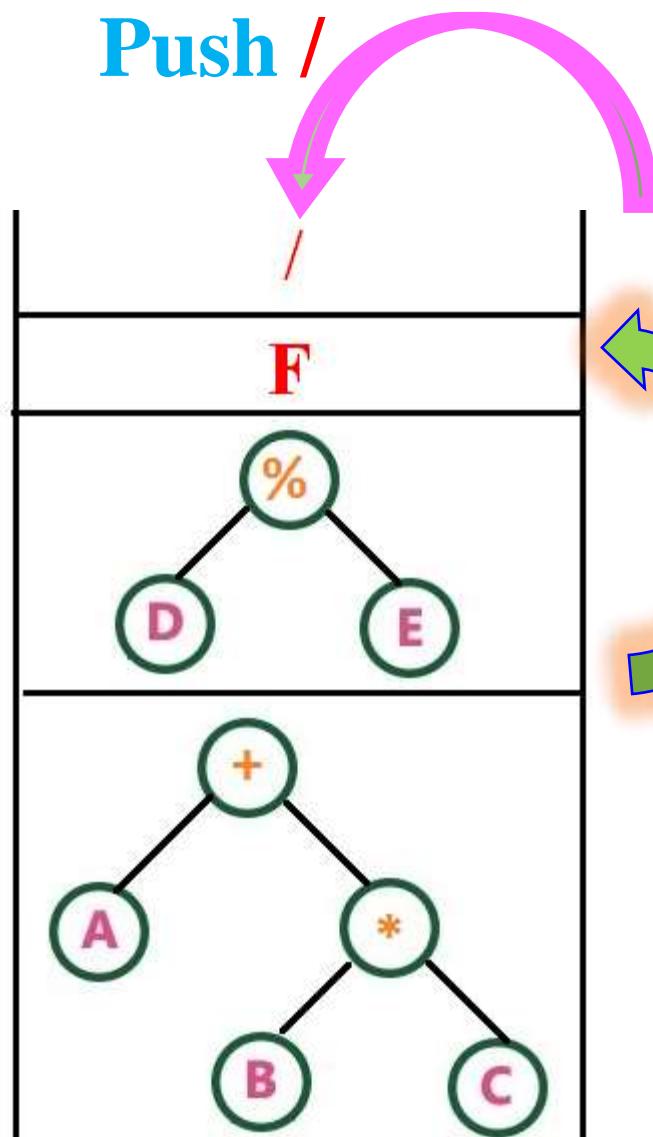
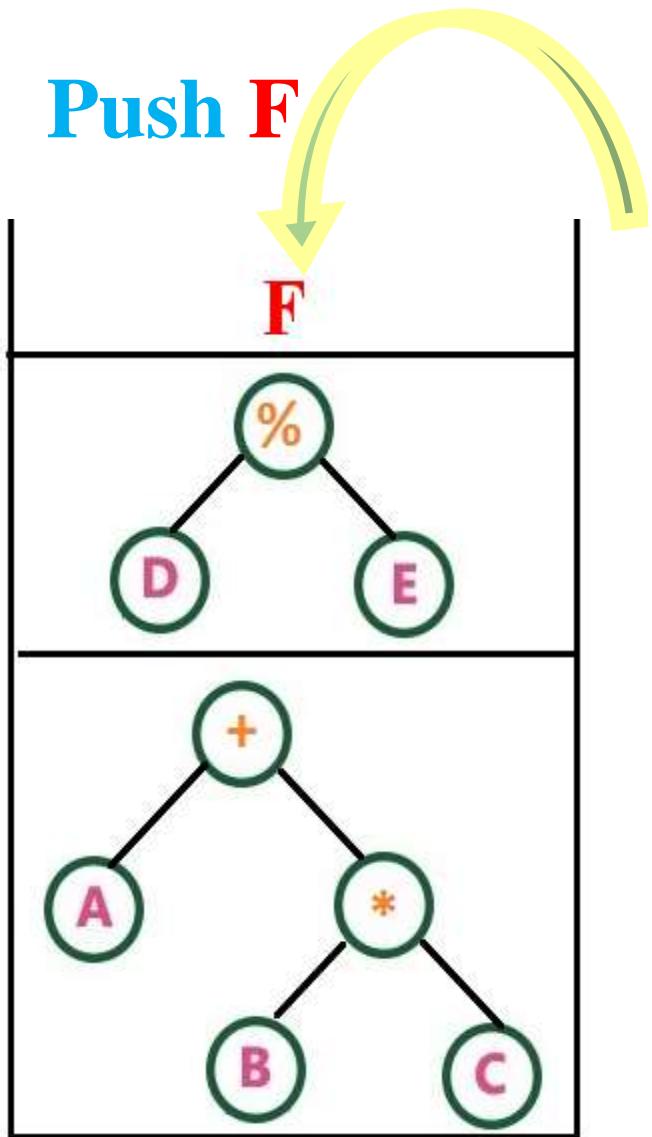


FROM THE POST FIX EXPRESSION **A B C * + D E % F / -**

A B C * + D E % F / -

Push /

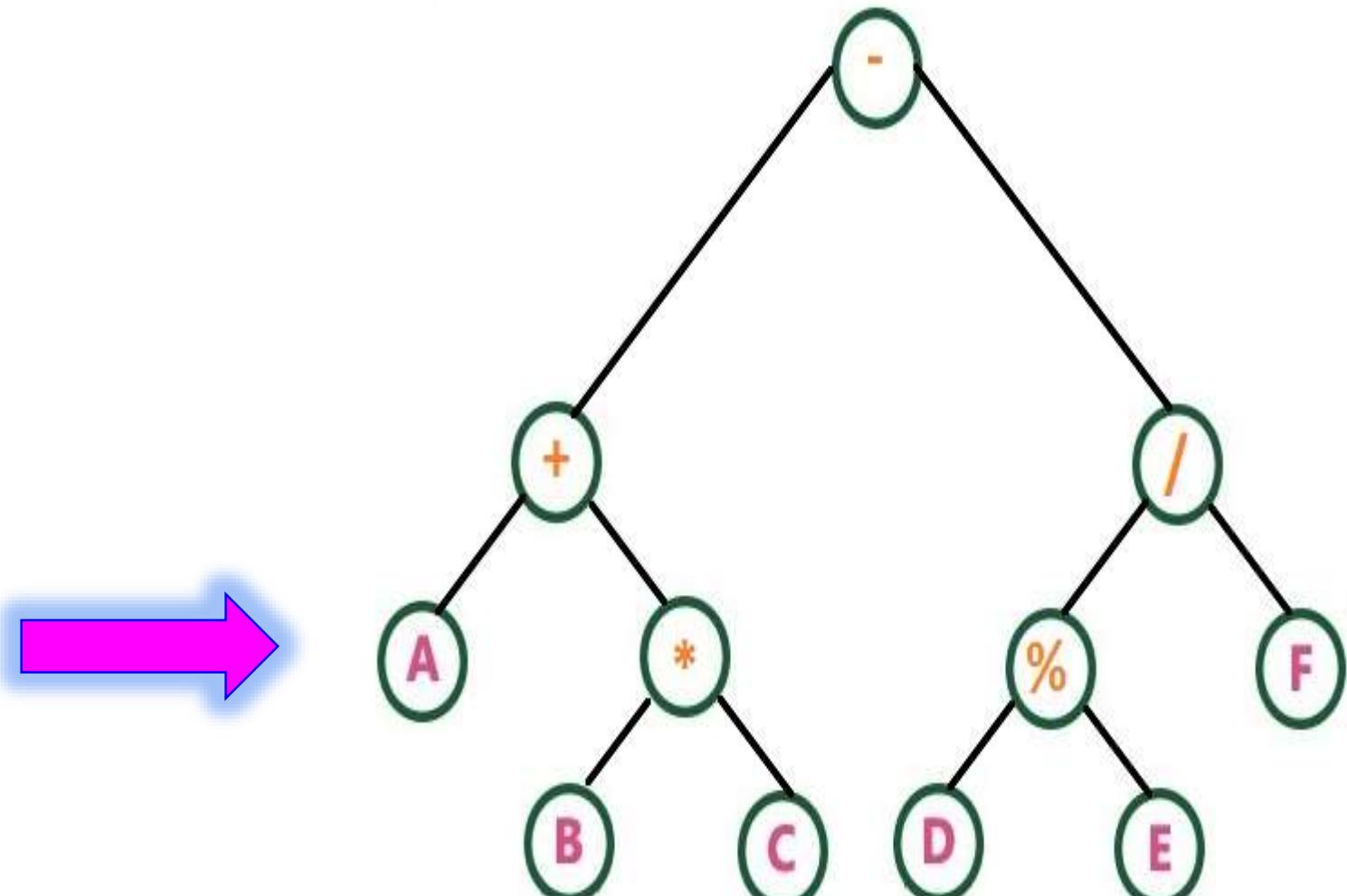
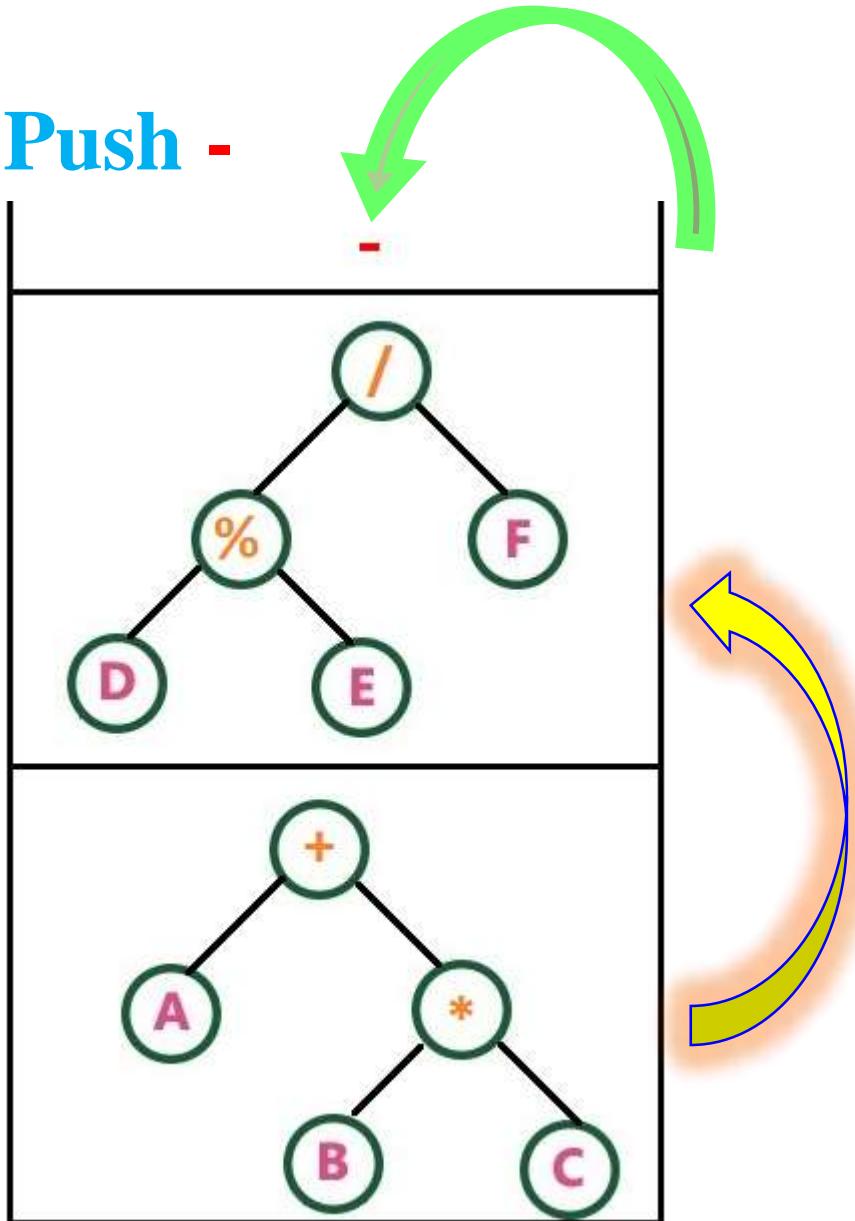
Push F



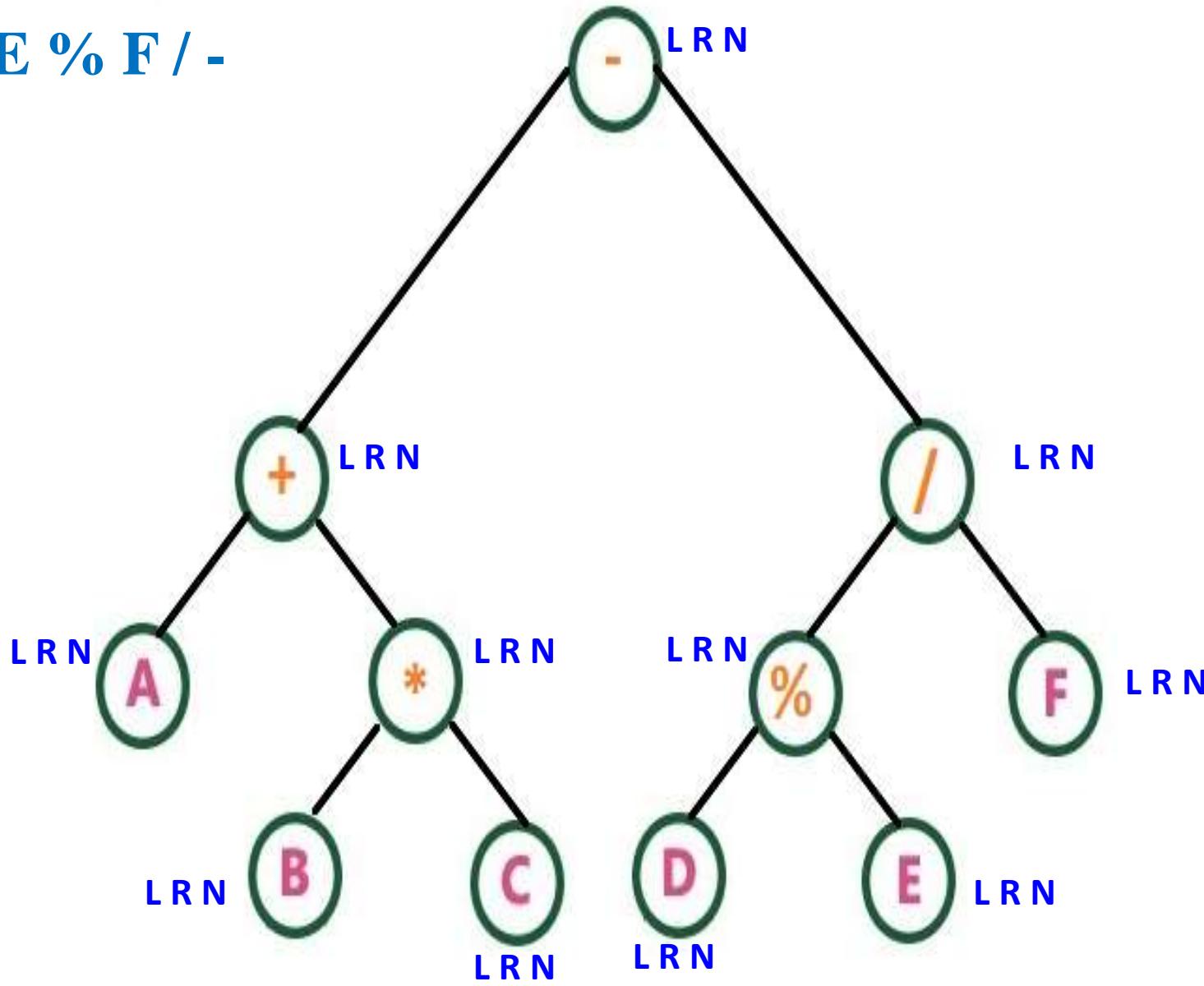
FROM THE POST FIX EXPRESSION **A B C * + D E % F / -**

A B C * + D E % F / -

Push -



A B C * + D E % F / -



**SAME WAY THE CONSTRUCTION OF
PRE-FIX EXPRESSION**

AVL TREES

AVL TREES
OR
HEIGHT BALANCED TREES

A V L

Adelson **Velsky** **Lindas**

GM Adelson – Velsky & EM Lindas

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honor of its inventors.

AVL Tree is also called as a **HEIGHT BALANCED TREE** ?

AVL TREES OR HEIGHT BALANCED TREES

- AVL is a Binary Tree/ Binary Search tree.
- AVL is a **SELF BALANCED** Tree.

What is **SELF BALANCING** ?

- ✓ Heights of left & right sub tree.
- ✓ Heights can be calculated with the help of **BALANCING FACTOR**.

AVL TREES

Balance factor:-

Balance factor = height Of Left Sub tree – height Of Right Sub tree.

- ✓ Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees.

- ✓ In an AVL tree, balance factor of every node is either.

-1, 0 or +1.

AVL TREES

-1 Heavy Right / Right Heavy Tree

0 Balanced Tree

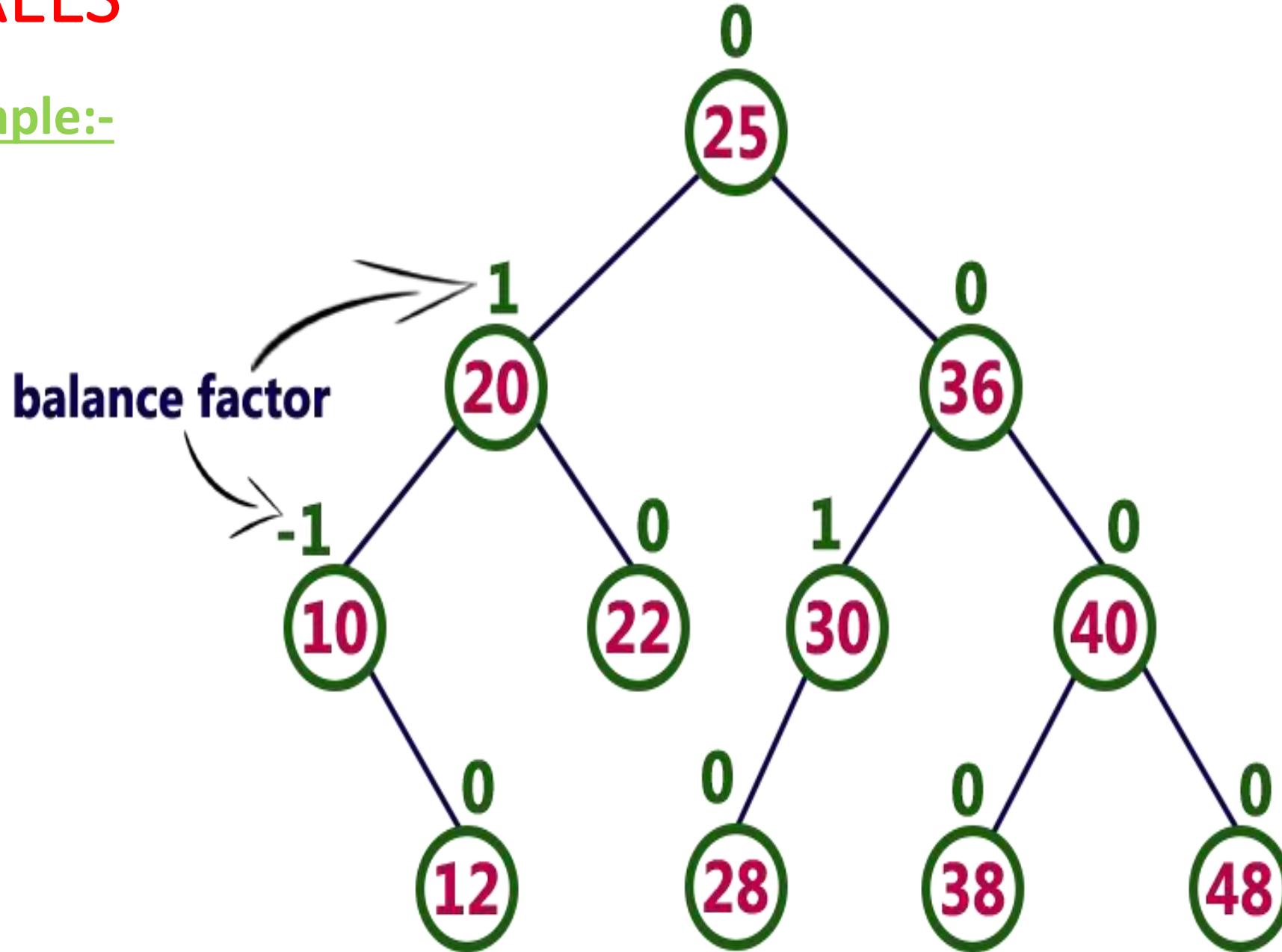
+1. Heavy Left / left Heavy Tree

OPERATIONS

- INSERTION
- SEARCHING
- DELETION

AVL TREES

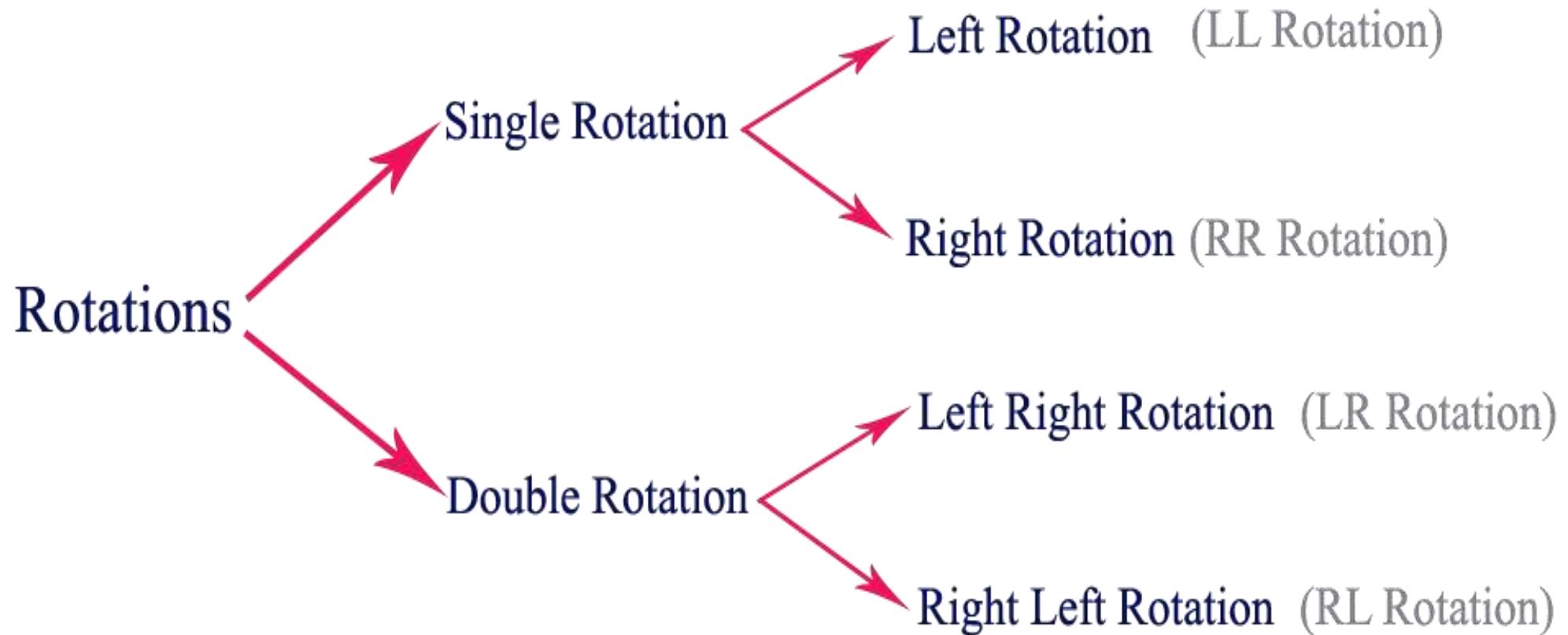
Example:-



AVL TREES

AVL Tree Rotations:-

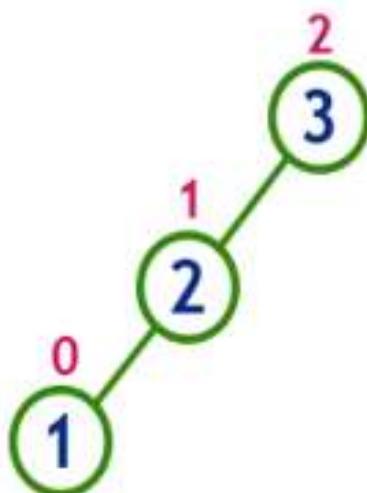
Rotation is the process of moving the nodes to either left or right to make tree balanced.



Single Left Rotation (LL Rotation)

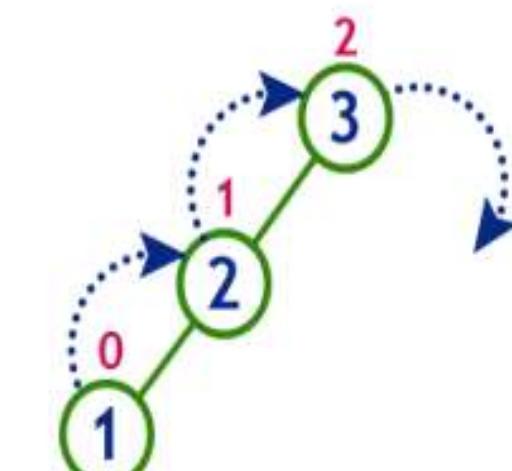
In **LL** Rotation every node moves one position to right from the current position. To understand **LL** Rotation.

insert 3, 2 and 1

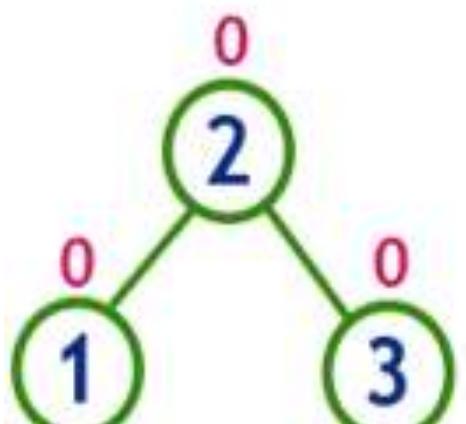


Tree is imbalanced

because node 3 has balance factor 2



To make balanced we use
Rotation which moves
nodes one position to right

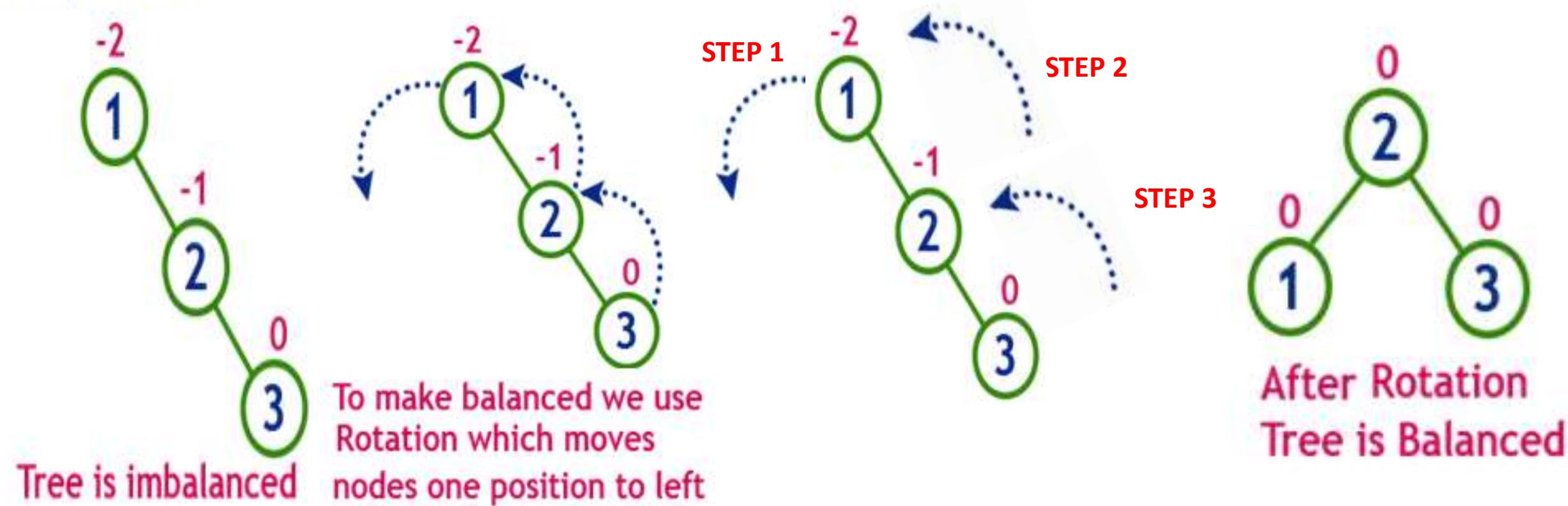


After Rotation
Tree is Balanced

Single Right Rotation (RR Rotation)

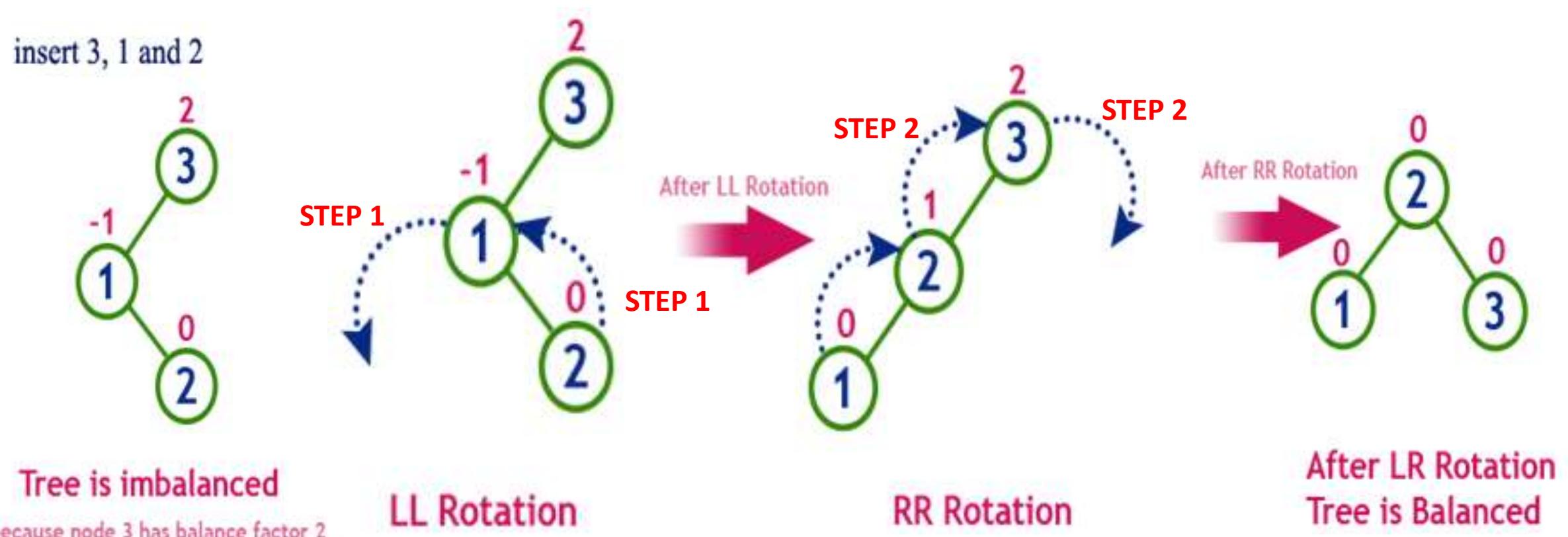
In **RR** Rotation every node moves one position to left from the current position. To understand **RR** Rotation.

insert 1, 2 and 3



Double Rotation left Right Rotation (LR Rotation)

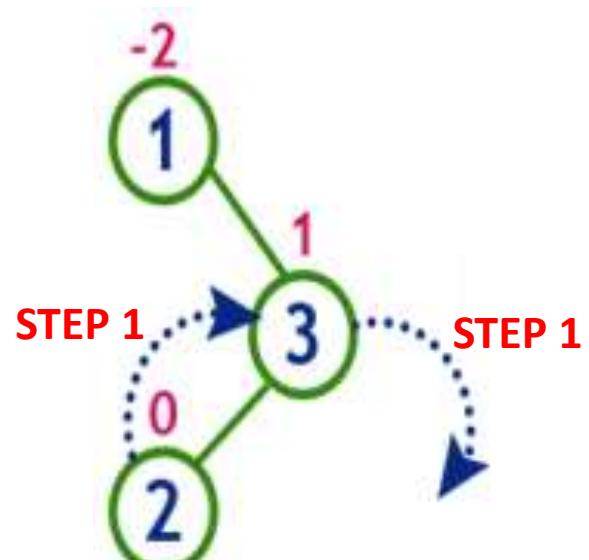
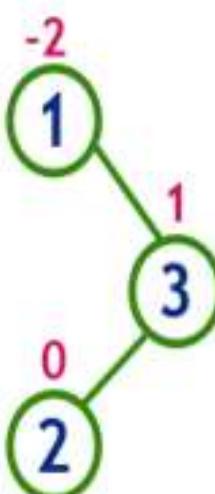
The LR Rotation is combination of single left rotation followed by single right rotation. In LR Rotation, first every node moves one position to left then one position to right from the current position.



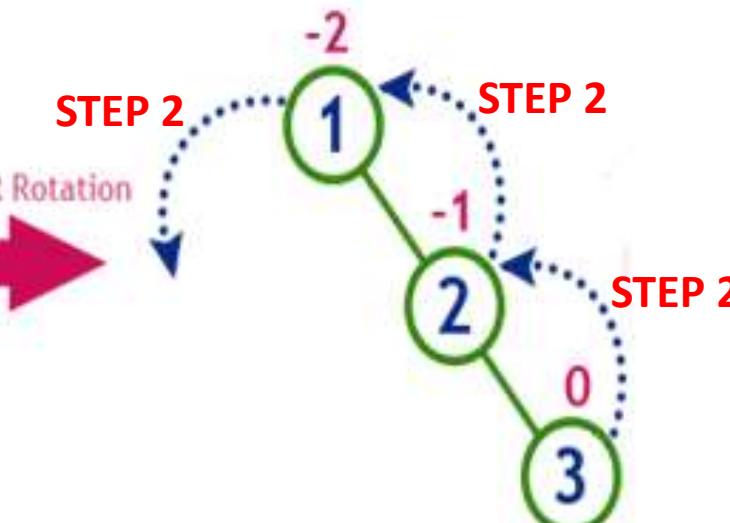
Right Left Rotation (RL Rotation)

The RL Rotation is combination of single right rotation followed by single left rotation. In RL Rotation, first every node moves one position to right then one position to left from the current position.

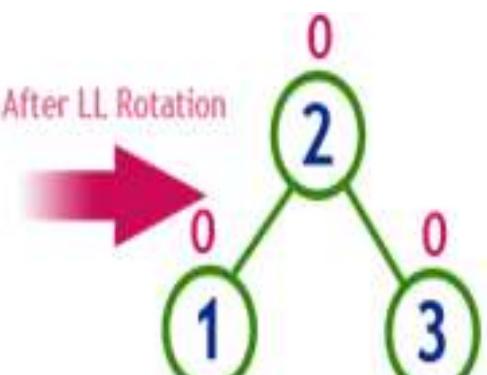
insert 1, 3 and 2



After RR Rotation



STEP 2
STEP 2
After LL Rotation



Tree is imbalanced
because node 1 has balance factor -2

RR Rotation

LL Rotation

After RL Rotation
Tree is Balanced

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with $O(\log n)$ time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
- **Step 5:** If search element is smaller, then continue the search process in left sub tree.
- **Step 6:** If search element is larger, then continue the search process in right sub tree.
- **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity.

In AVL Tree, new node is always inserted as a leaf node.

- **Step 1:** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2:** After insertion, check the **Balance Factor** of every node.
- **Step 3:** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4:** If the **Balance Factor** of any node is other than **0 or 1 or -1** then tree is said to be imbalanced. Then perform the suitable **Rotation** to make it balanced. And go for next operation

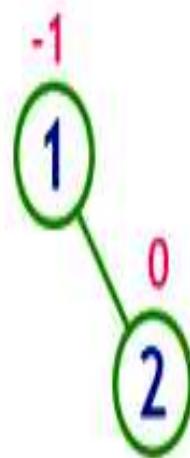
Construct an AVL Tree by inserting numbers from 1 to 8

insert 1



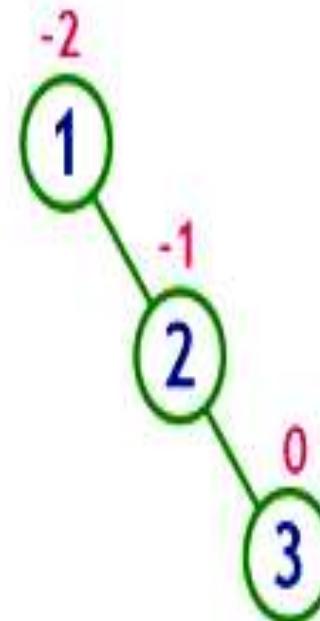
Tree is balanced

insert 2

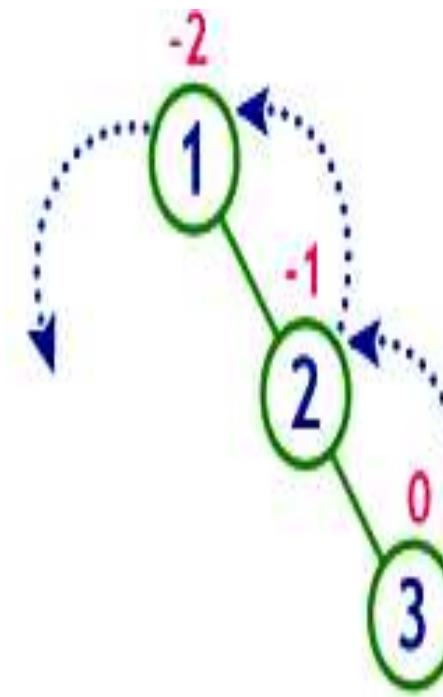


Tree is balanced

insert 3

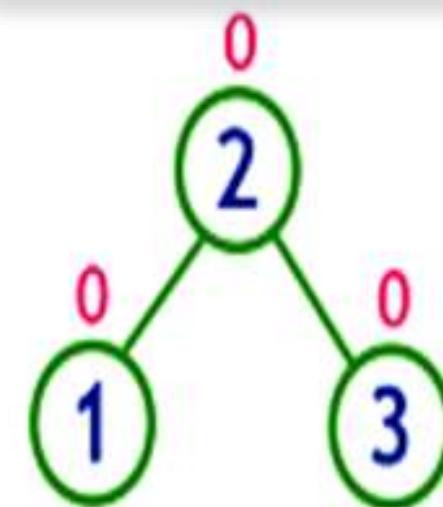


Tree is imbalanced



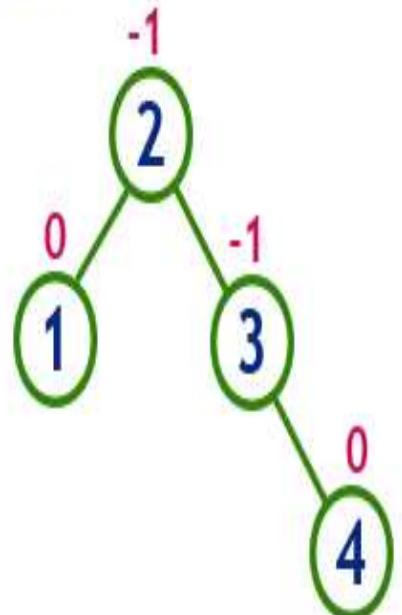
LL Rotation

After RR Rotation at 1



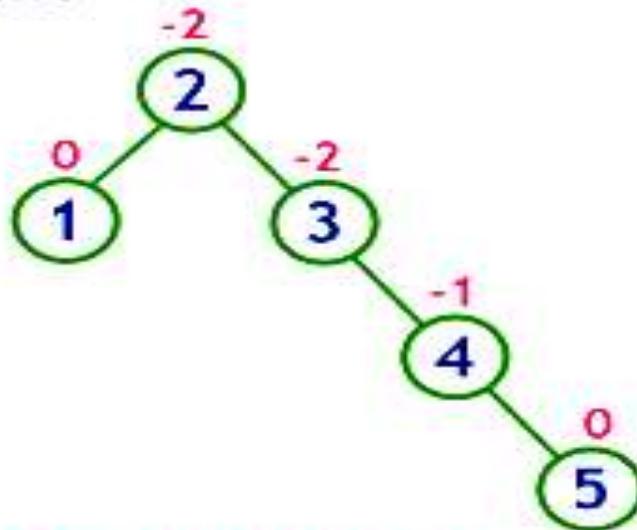
Tree is balanced

insert 4

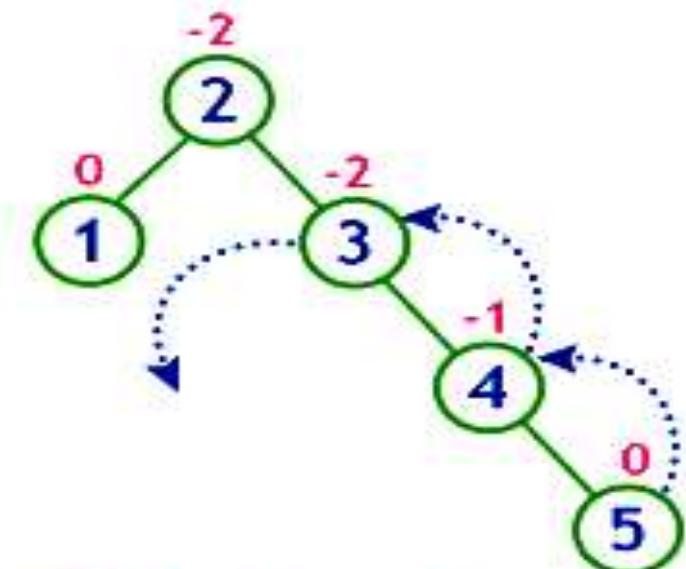


Tree is balanced

insert 5

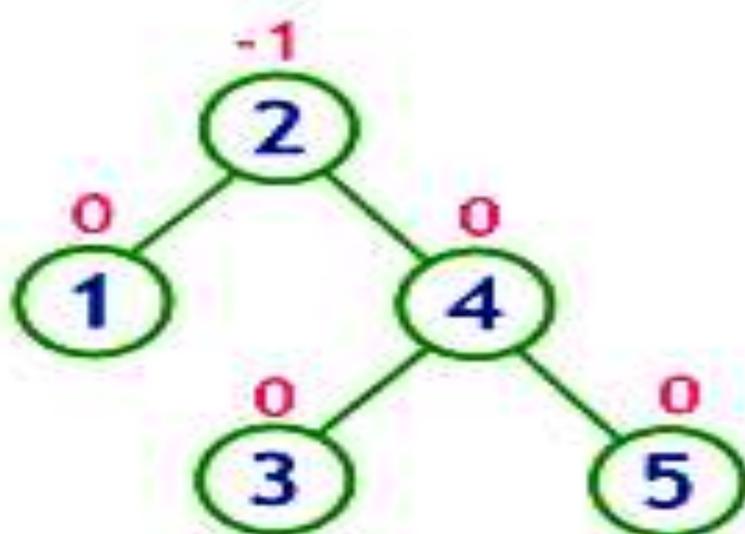


Tree is imbalanced



RR Rotation at 3

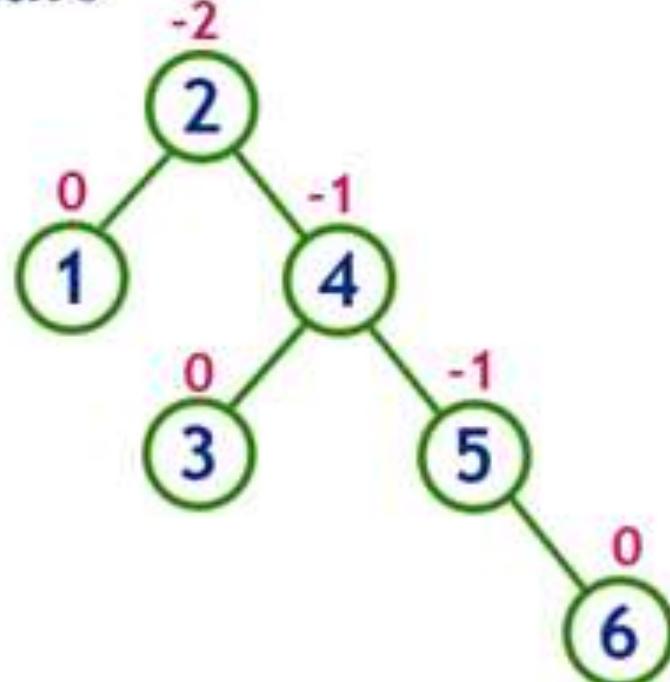
After RR Rotation at 3



Tree is balanced

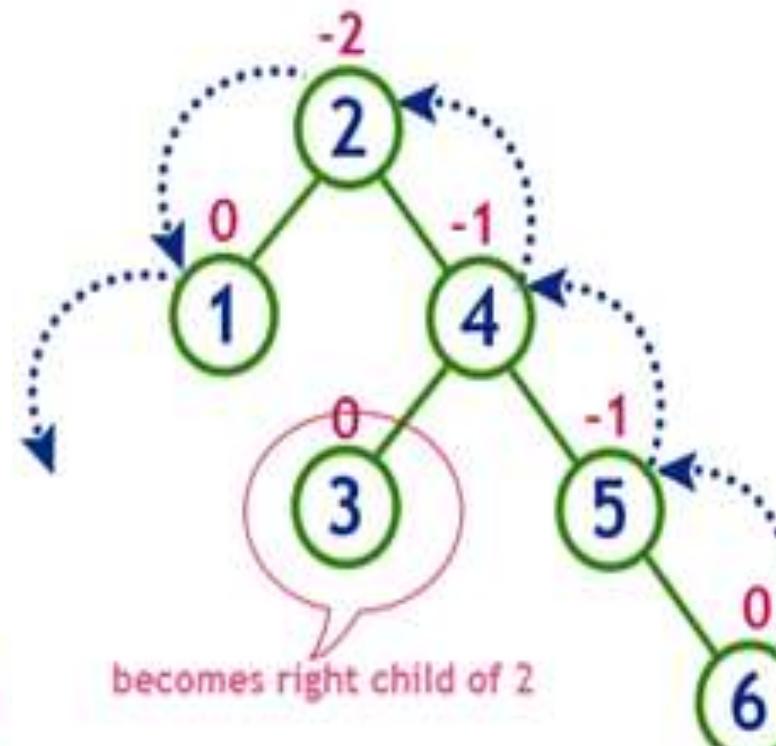
Apply RR Rotation at 2

insert 6

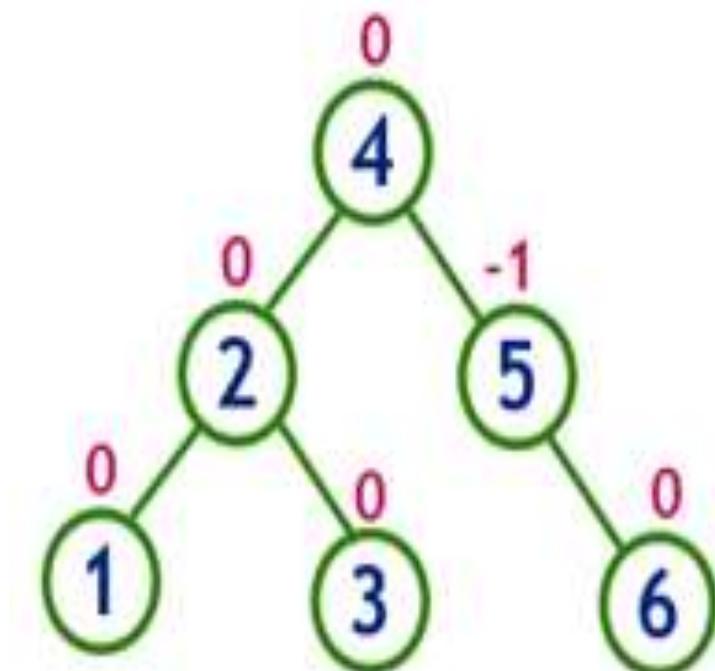


Tree is imbalanced

After RR Rotation at 2



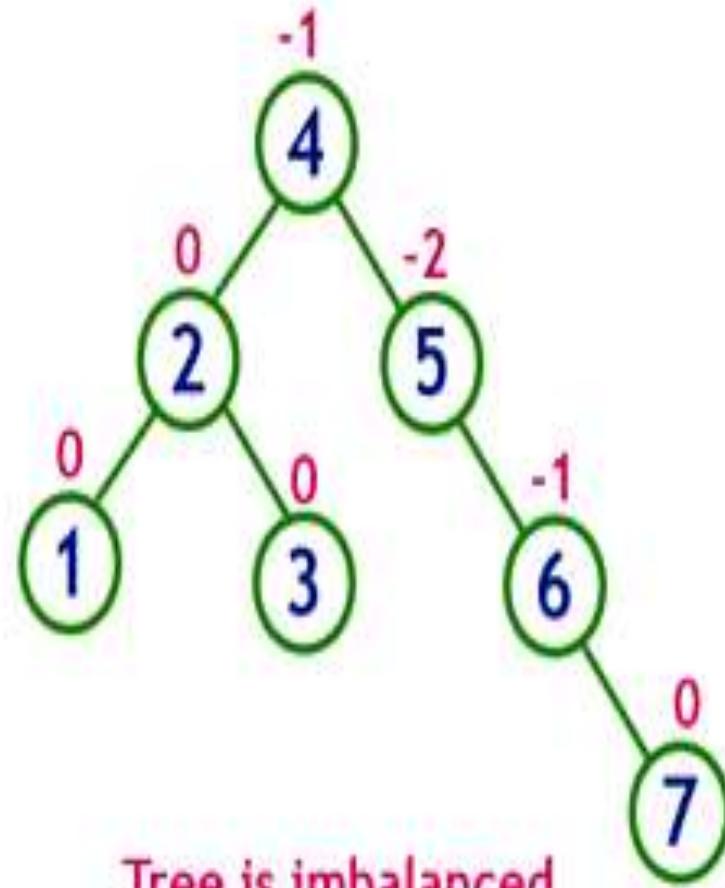
RR Rotation at 2



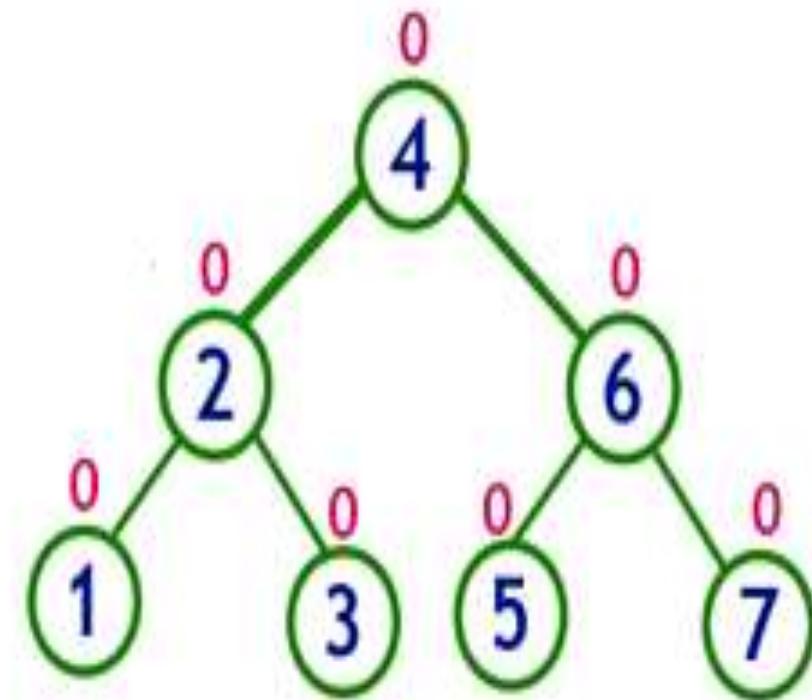
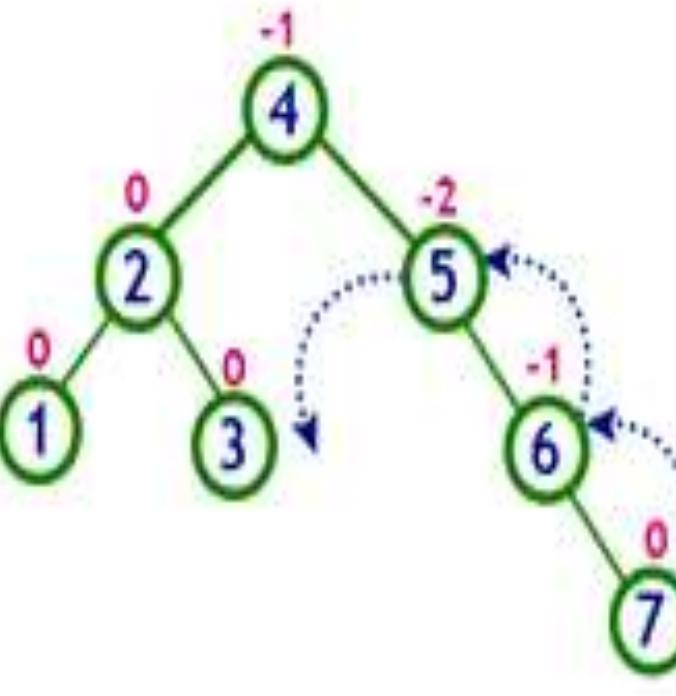
Tree is balanced

insert 7

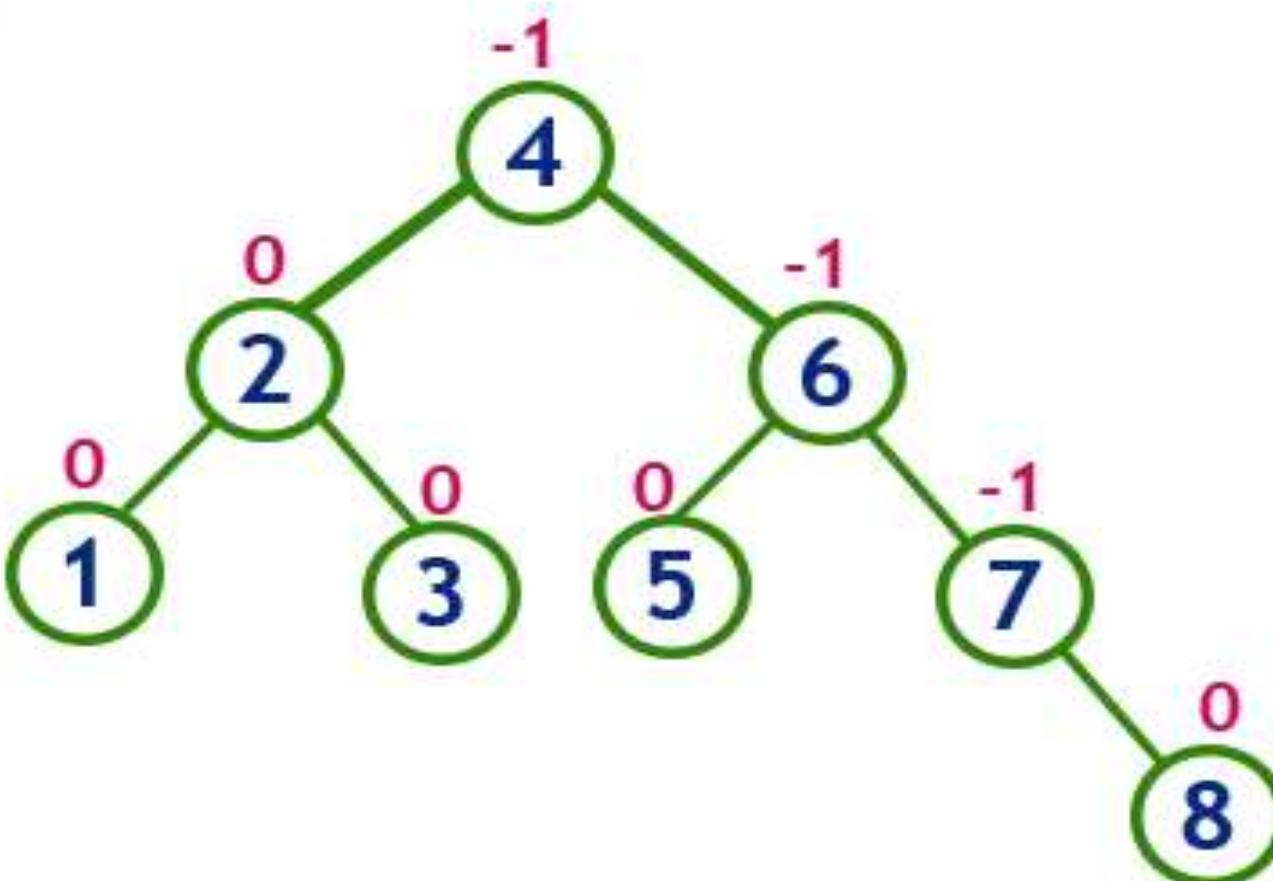
Apply RR Rotation at 5



After RR Rotation at 5



insert 8



Tree is balanced

Deletion operation on an AVL Tree

- ✓ In an AVL Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Balance Factor condition.
- ✓ If the tree is balanced after deletion then go for next operation otherwise perform the suitable rotation to make the tree Balanced.

B-TREES

B - Trees

In a binary search tree, AVL Tree, Red-Black tree etc., every node can have only one value (key) and maximum of two children but there is another type of search tree called B-Tree in which a node can store more than one value (key) and it can have more than two children.

B-Tree was developed in the year of 1972 by **Bayer and McCreight** with the name **Height Balanced m-way Search Tree**. Later it was named as B-Tree.

B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node.

Generalization of BST in which a node can have more than one key value & more than two children.

Here, number of keys in a node and number of children for a node is depend on the order of the B-Tree. **Every B-Tree has order.**

B - Trees

B-Tree of Order m has the following properties...

- **Property 1** - All the **leaf nodes** must be **at same level**.
- **Property 2** - All nodes except root must have at least $[m/2]-1$ keys and maximum of $m-1$ keys.
- **Property 3** - All non leaf nodes except root (i.e. all internal nodes) must have at least $m/2$ children.
- **Property 4** - If the root node is a non leaf node, then it must have **at least 2** children.
- **Property 5** - A non leaf node with $n-1$ keys must have n number of children.
- **Property 6** - All the **key values within a node** must be in **Ascending Order**.

Operations on a B-Tree

- Search
- Insertion
- Deletion

HOW TO FIND THE ORDER

Maximum Number Of Childs = [M]

Maximum Number Of Keys = [M-1]

Minimum Number Of Childs =[M/2]-1

Minimum Number Of keys =[M/2]

For order 1

$$[m] = 1$$

$$[m-1] = 1 - 1 = 0$$

$$[m/2]-1 = 0.5 = 1 - 1 = 0$$

$$[m/2] = 0.5 = 1$$

For order 2

$$[m] = 2$$

$$[m-1] = 1$$

$$[m/2]-1 = 1 - 1 = 0$$

$$[m/2] = 1$$

For order 3

$$[m] = 3$$

$$[m-1] = 2$$

$$[m/2]-1 = 1.5 = 1.5 - 1 = 0.5 = 1$$

$$[m/2] = 1.5 = 2$$

Insertion Operation in B-Tree

- In a B-Tree, the new element must be added only at leaf node. That means, always the new key Value is attached to leaf node only. The insertion operation is performed as follows...
- **Step 1:** Check whether tree is Empty.
- **Step 2:** If tree is **Empty**, then create a new node with new key value and insert into the tree as a root node.
- **Step 3:** If tree is **Not Empty**, then find a leaf node to which the new key value can be added using Binary Search Tree logic.
- **Step 4:** If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.
- **Step 5:** If that leaf node is already full, then **split** that leaf node by sending middle value to its parent node. Repeat the same until sending value is fixed into a node.
- **Step 6:** If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

Construct a B-Tree of Order 3 by inserting numbers from 1 to 10.

Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

1	
---	--

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

1	2
---	---

Example

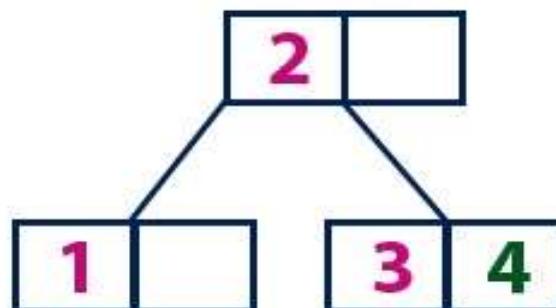
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't have an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't have a parent. So, this middle value becomes a new root node for the tree.



insert(4)

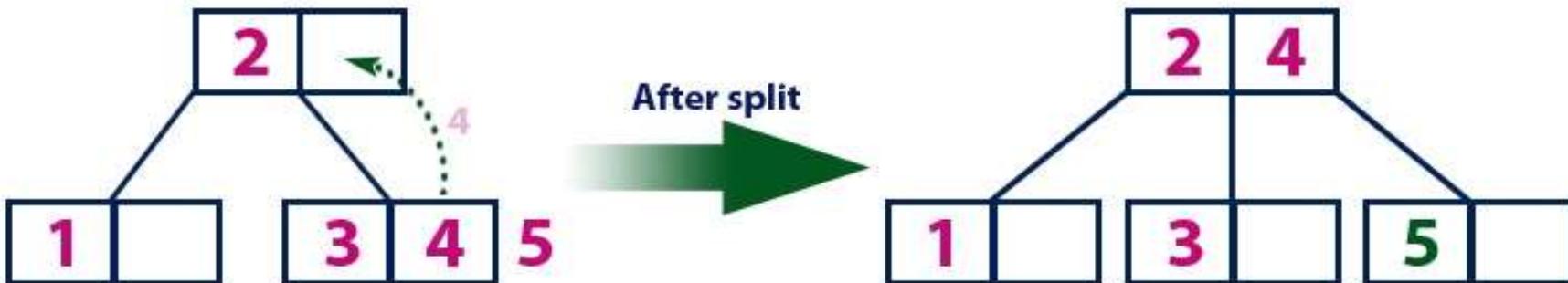
Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



Example

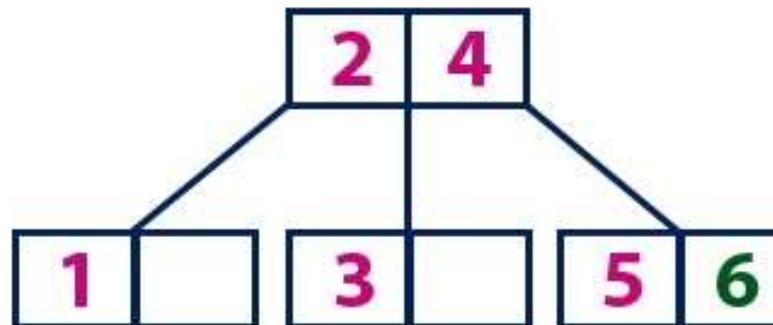
insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.



insert(6)

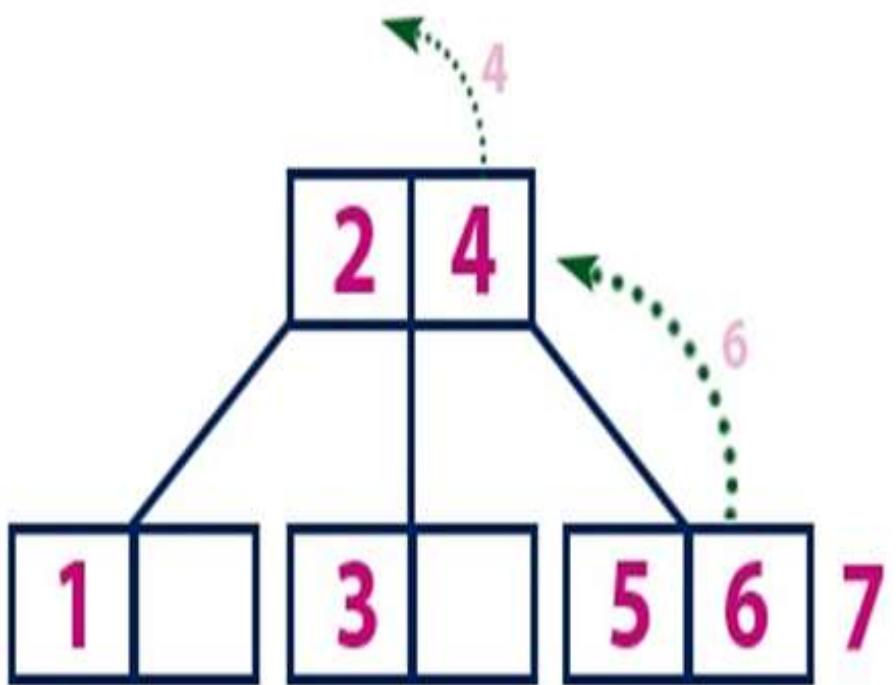
Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



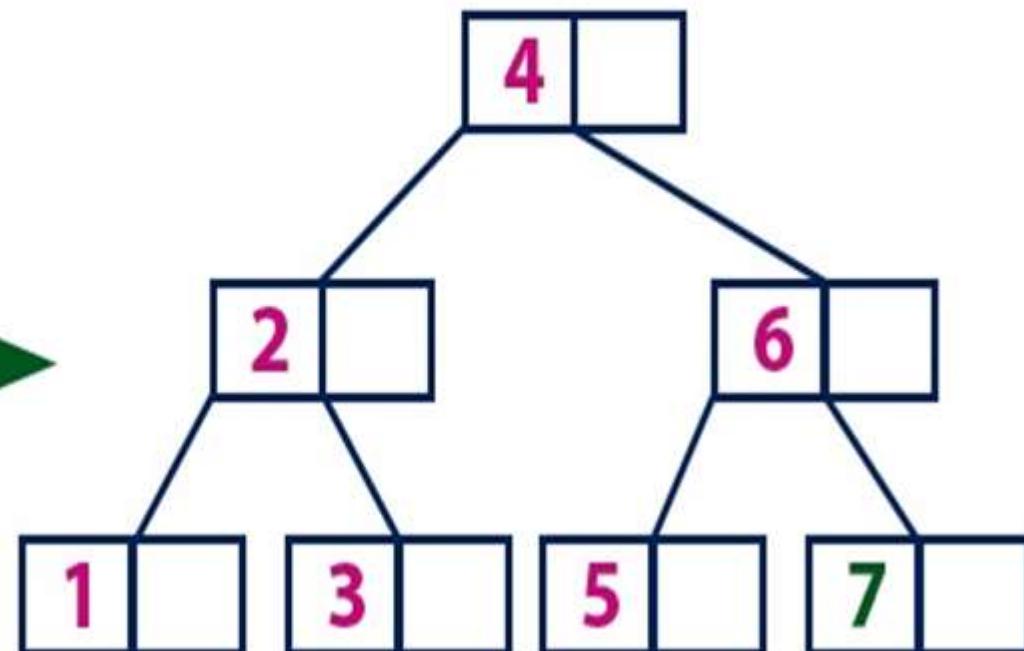
Example

insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



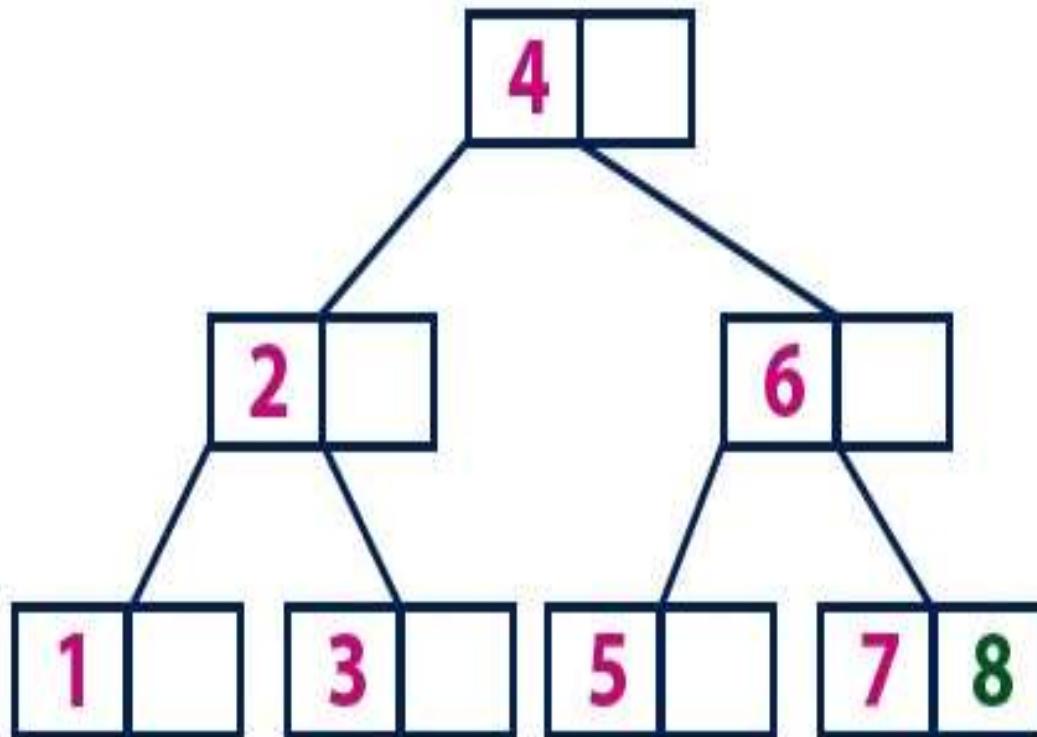
After split



Example

insert(8)

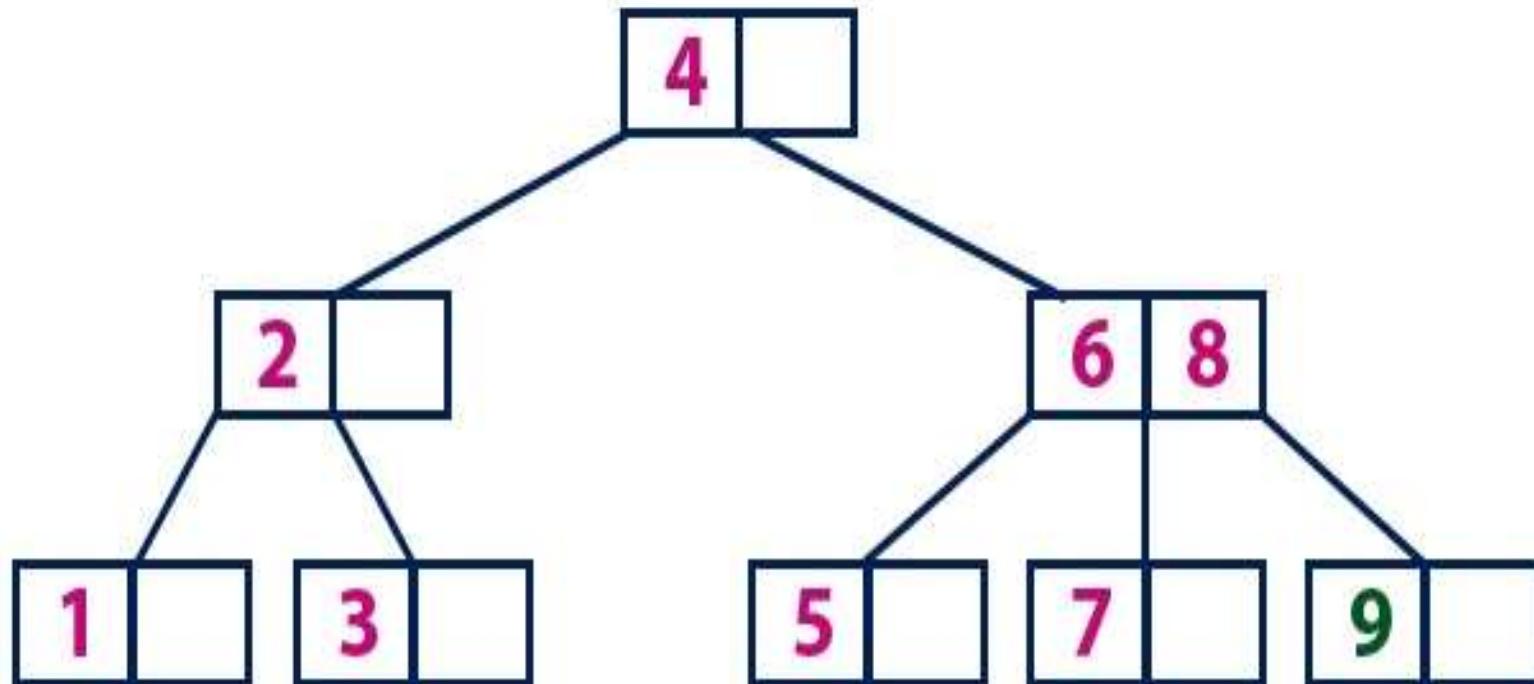
Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



Example

insert(9)

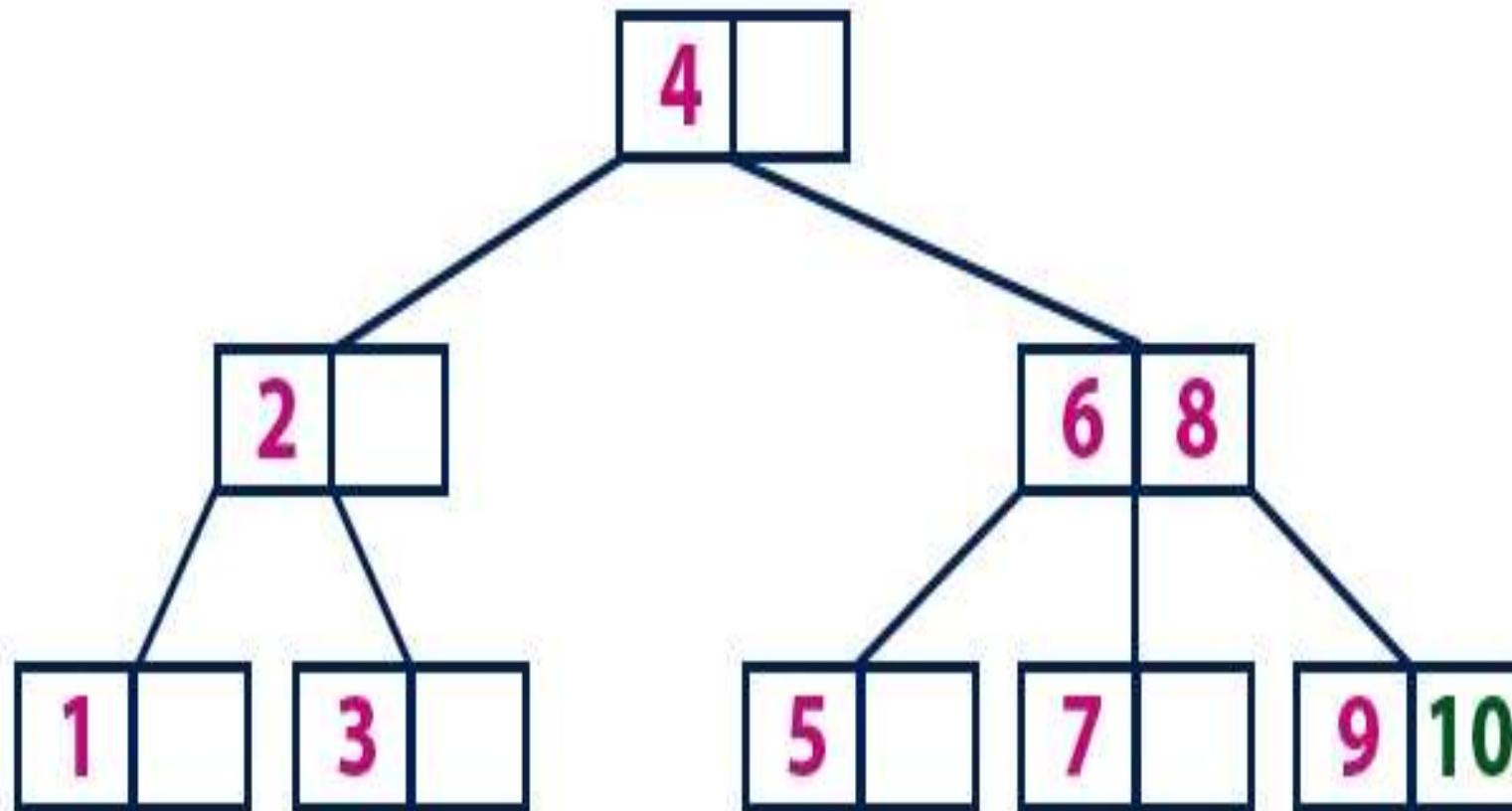
Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



Example

insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8'. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



For order 4

$$[m] = 4$$

$$[m-1] = 3$$

$$[m/2]-1 = 2 = 2-1=1$$

$$[m/2] = 2$$

Search Operation in B-Tree

- In a B-Tree, the search operation is similar to that of Binary Search Tree. In a Binary search tree, the search process starts from the root node and every time we make a 2-way decision (we go to either left sub tree or right sub tree).
- In B-Tree also search process starts from the root node but every time we make n-way decision where n is the total number of children that node has.
- In a B-Tree, the search operation is performed with $O(\log n)$ time complexity.
- The search operation is performed as follows...

Search Operation in B-Tree

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with first key value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that key value.
- **Step 5:** If search element is smaller, then continue the search process in left sub tree.
- **Step 6:** If search element is larger, then compare with next key value in the same node and repeat step 3, 4, 5 and 6 until we found exact match or comparison completed with last key value in a leaf node.

Red Black Tree



Red Black Tree

Red - Black Tree is another variant of Binary Search Tree in which every node is colored either RED or BLACK. We can define a Red Black Tree as follows...

- In a Red Black Tree the color of a node is decided based on the Red Black Tree properties. Every Red Black Tree has the following properties.

Properties of Red Black Tree:-

Property 1: Red - Black Tree must be a Binary Search Tree.

Property 2: The ROOT node must colored BLACK.

Property 3: The children of Red colored node must colored BLACK. (There should not be two consecutive RED nodes).

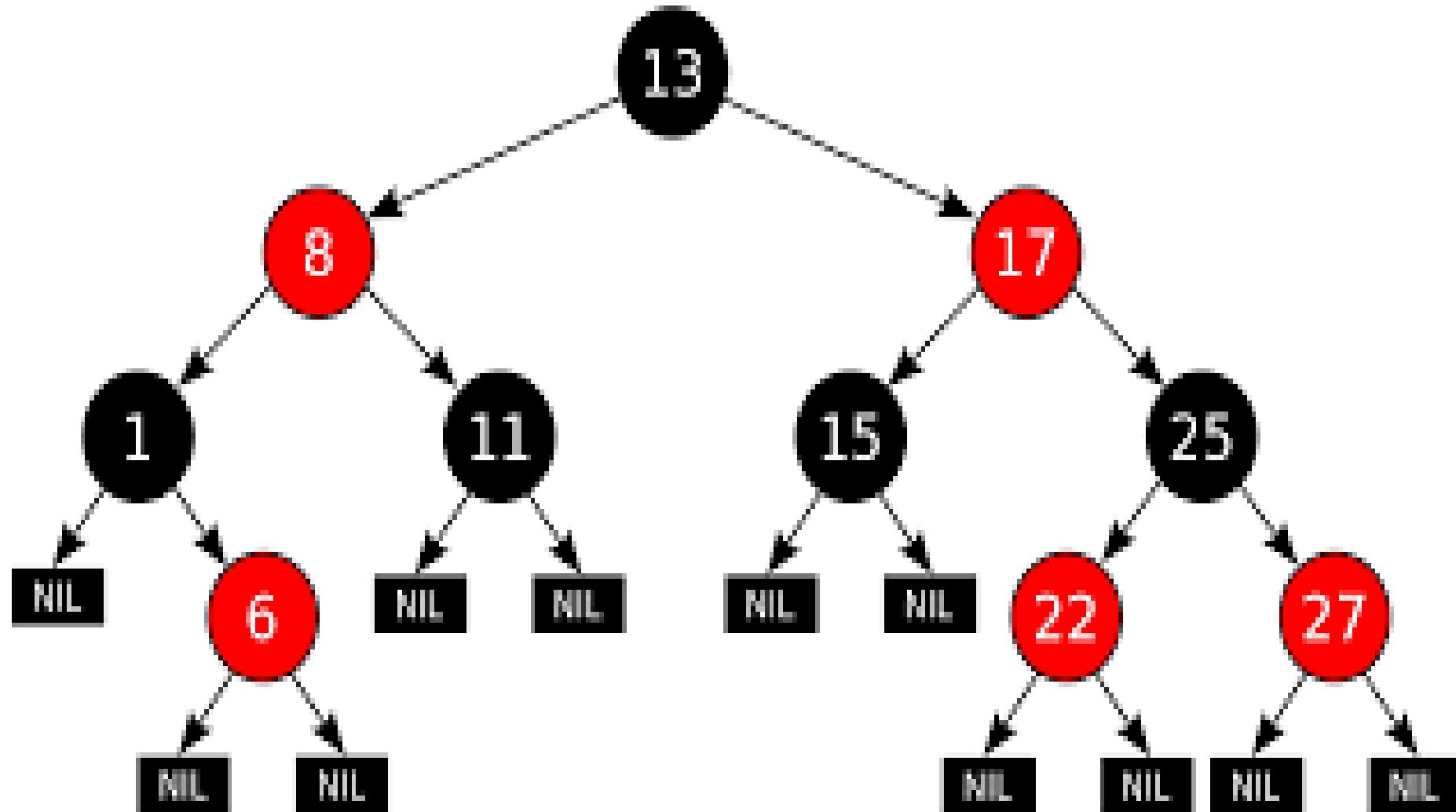
Property 4: In all the paths of the tree there must be same number of BLACK colored nodes.

Property 5: Every new node must inserted with RED color.

Property 6: Every leaf (e.i. NULL node) must colored BLACK.

Red Black Tree

✓ Example



Red Black Tree

Insertion into RED BLACK Tree:-

- In a Red Black Tree, every new node must be inserted with color **RED**. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. **But it is inserted with a color property.**
- After every insertion operation, **we need to check all the properties of Red Black Tree**. If all the properties are satisfied then we go to next operation otherwise we need to perform following operation to make it Red Black Tree.

Red Black Tree

✓ OPERATIONS OF RED BLACK TREE

- Recolor
- Rotation followed by Recolor
- Deletion

Properties of Red-Black Trees

A red-black tree is a binary search tree in which every node has a color which is either red or black.

1. The color of a node is either red or black.
2. The color of the root node is always black.
3. All leaf nodes are black.
4. Every red node has both the children colored in black.
5. Every simple path from a given node to any of its leaf nodes has an equal number of black nodes.

Red Black Tree

- ✓ The insertion operation in Red Black tree is performed using following steps...
- **Step 1:** Check whether tree is Empty.
- **Step 2:** If tree is Empty then insert the **new Node** as Root node with color **Black** and exit from the operation.
- **Step 3:** If tree is not Empty then insert the new Node as a leaf node with Red color.
- **Step 4:** If the parent of new Node is Black then exit from the operation.
- **Step 5:** If the parent of new Node is Red then check the color of parent node's sibling of new Node.
- **Step 6:** If it is Black or NULL node then make a suitable Rotation and Recolor it.
- **Step 7:** If it is Red colored node then perform Recolor and Recheck it. Repeat the same until tree becomes Red Black Tree.

Red Black Tree Example

Create a RED BLACK Tree by inserting following sequence of number
8, 18, 5, 15, 17, 25, 40 & 80.

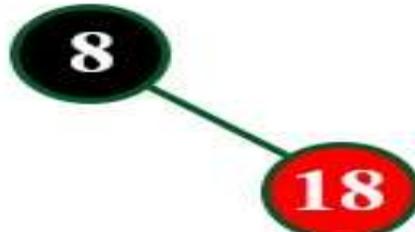
insert (8)

Tree is Empty. So insert newNode as Root node with black color.



insert (18)

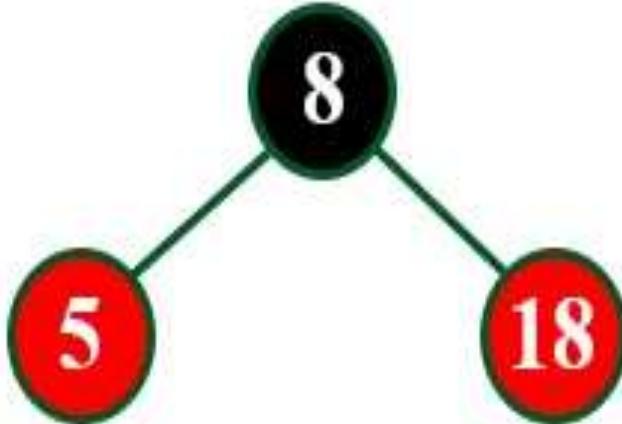
Tree is not Empty. So insert newNode with red color.



Red Black Tree

insert (5)

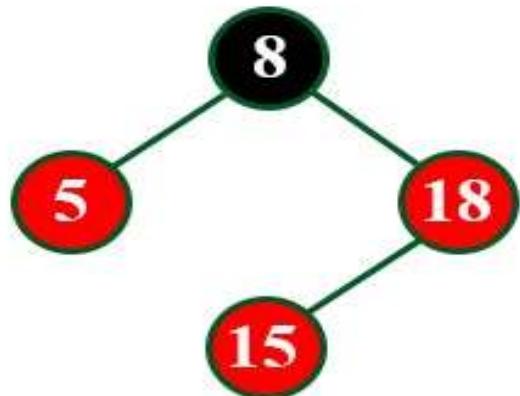
Tree is not Empty. So insert newNode with red color.



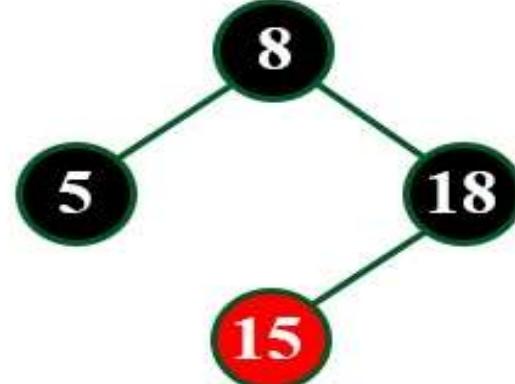
Red Black Tree

insert (15)

Tree is not Empty. So insert newNode with red color.



After RECOLOR



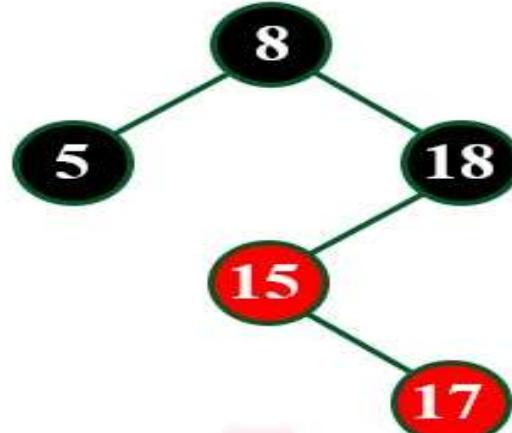
Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

After Recolor operation, the tree is satisfying all Red Black Tree properties.

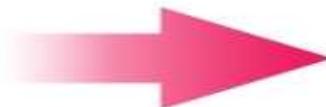
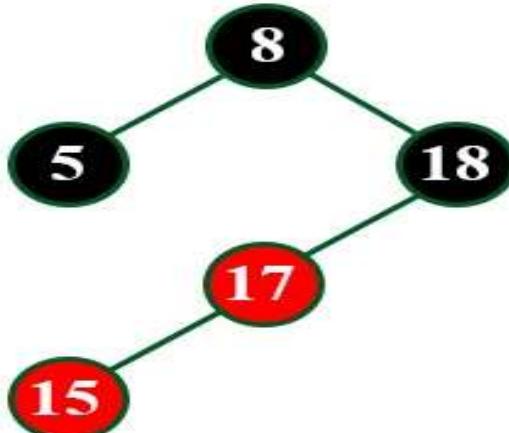
Red Black Tree

insert(17)

Tree is not Empty. So insert newNode with red color.

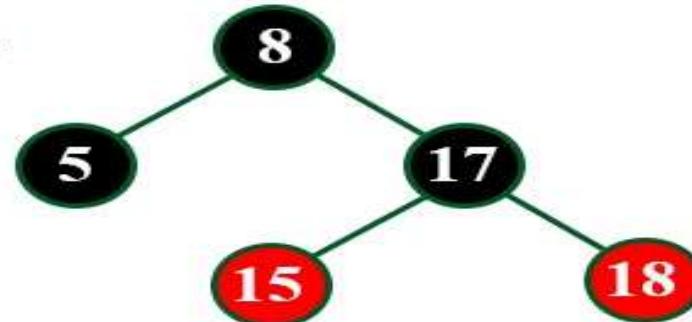


After Left Rotation



After Right Rotation & Recolor

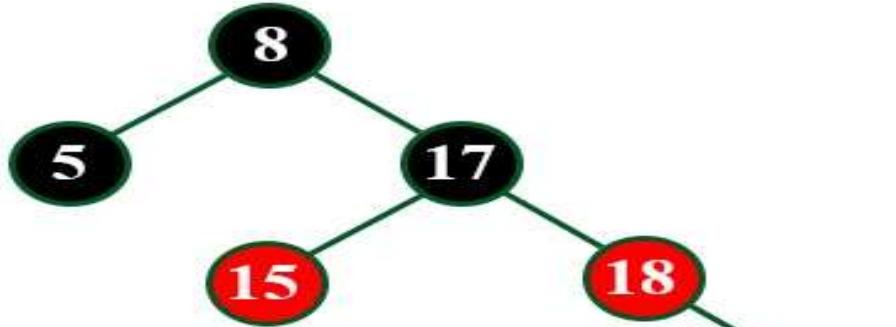
Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.



Red Black Tree

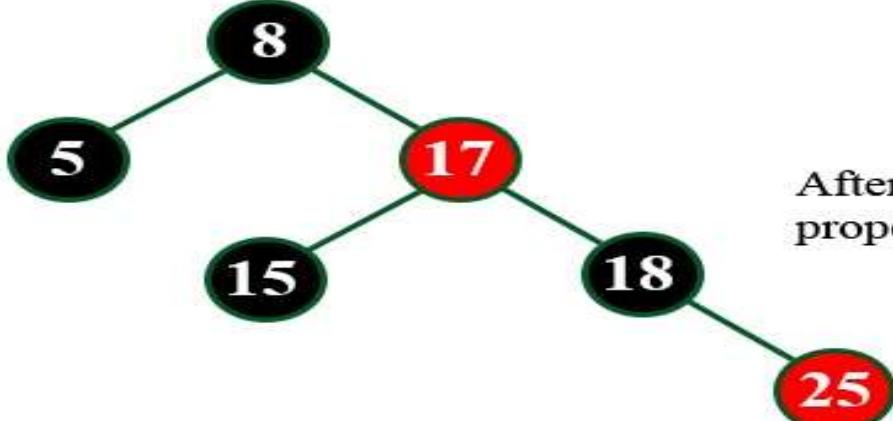
insert (25)

Tree is not Empty. So insert newNode with red color.



After Recolor

Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

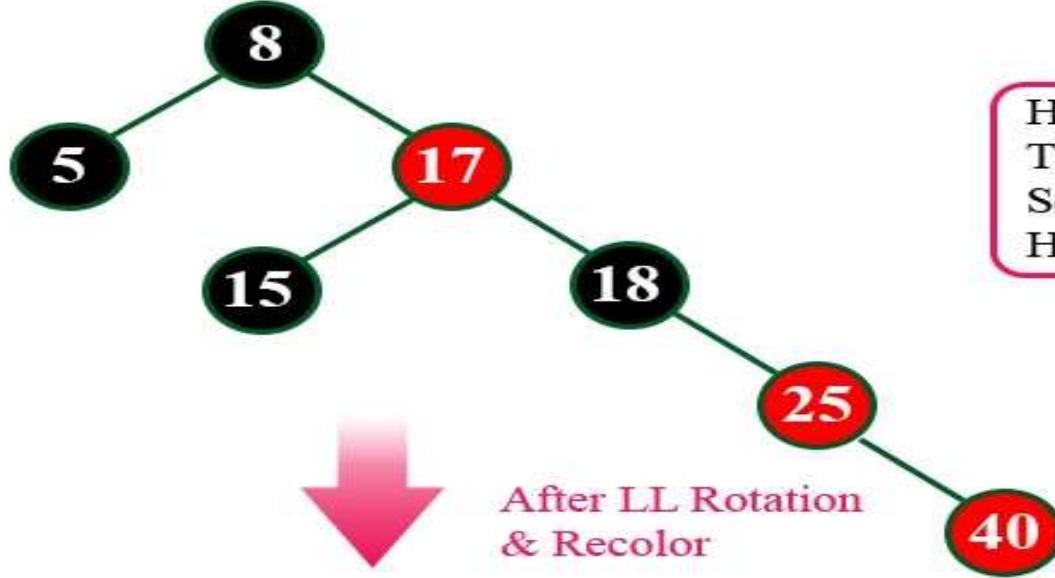


After Recolor operation, the tree is satisfying all Red Black Tree properties.

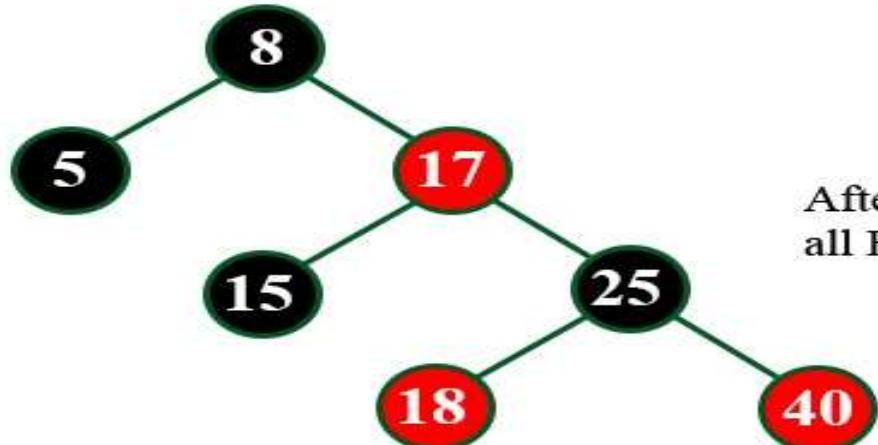
Red Black Tree

insert (40)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL.
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

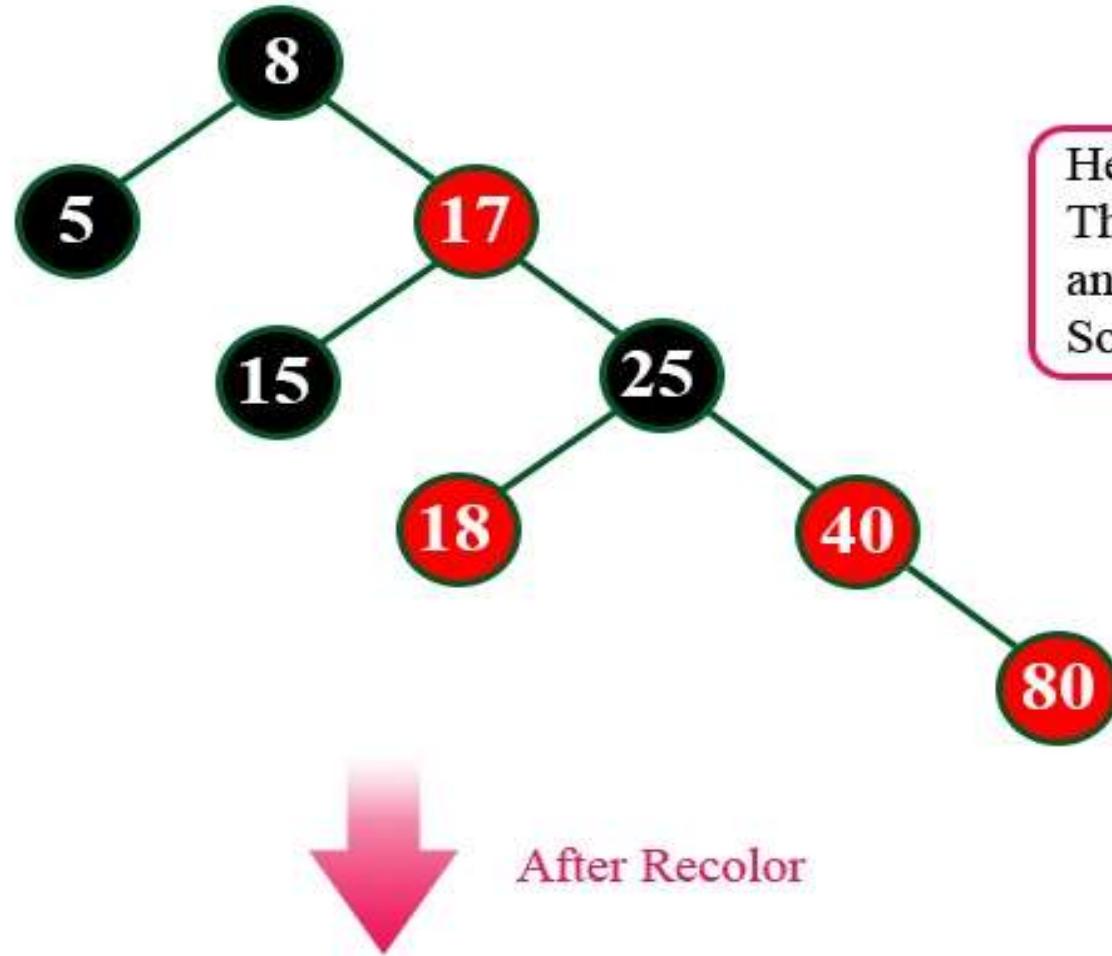


After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

Red Black Tree

insert (80)

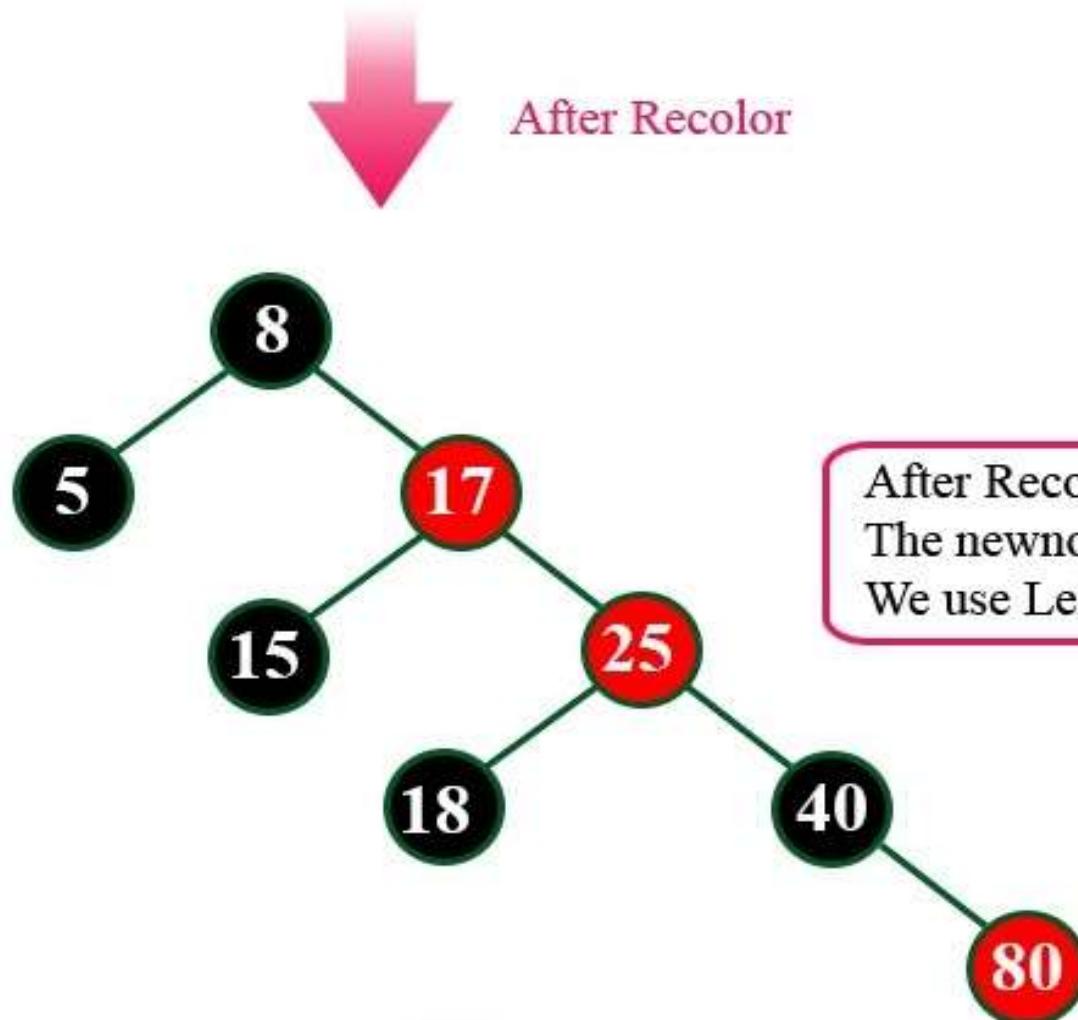
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor

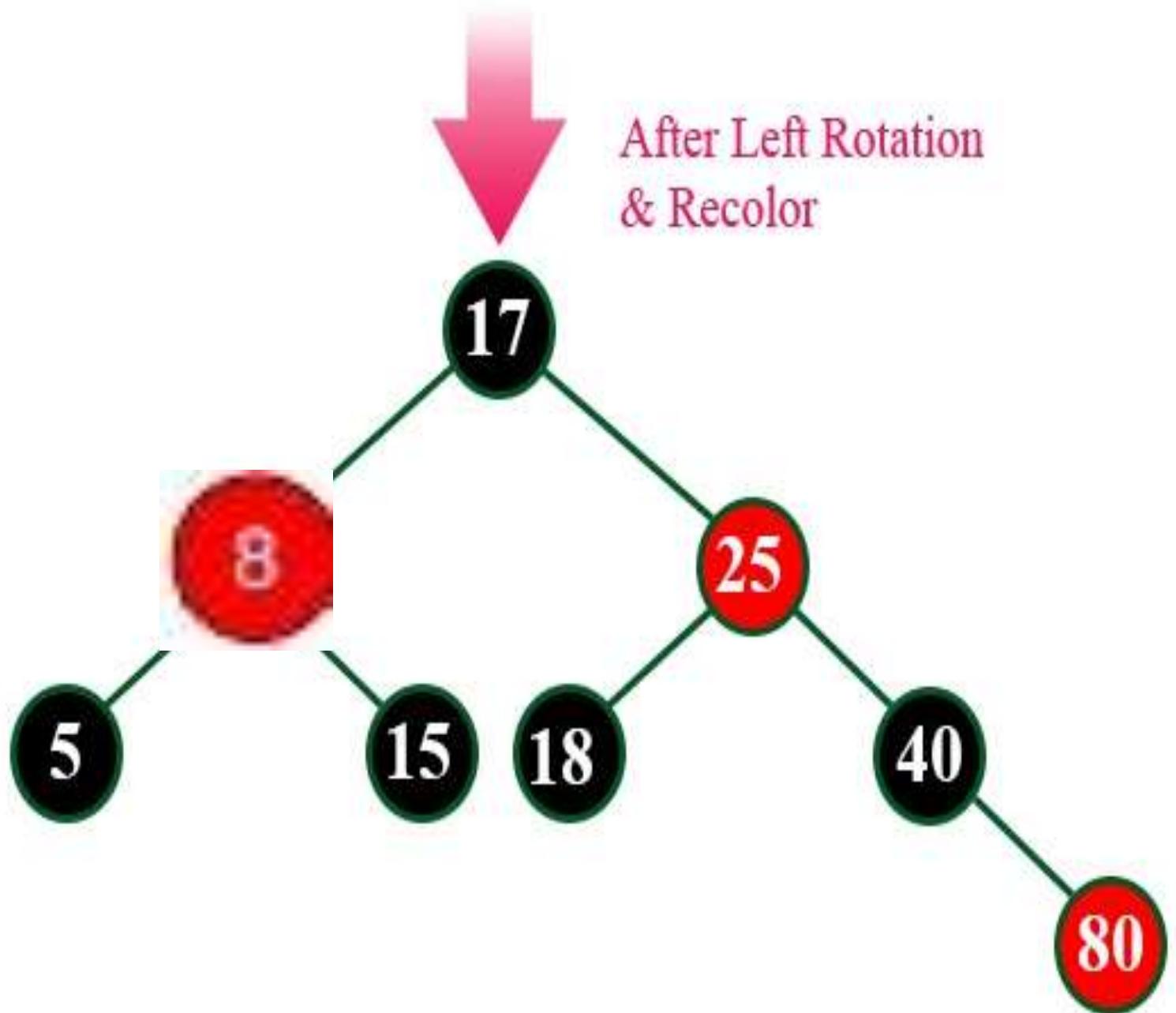
Red Black Tree



After Recolor

After Recolor again there are two consecutive Red nodes (17 & 25). The newnode's parent sibling color is Black. So we need Rotation. We use Left Rotation & Recolor.

Red B'



Red Black Tree

✓ Deletion Operation in Red Black Tree

- In a Red Black Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Red Black Tree properties.
- If any of the property is violated then make suitable operation like Recolor or Rotation & Recolor.

HEAP TREE

HEEP TREE

- Heap data structure is a specialized binary tree-based data structure.
- Heap is a binary tree with special characteristics.
- In a heap data structure, nodes are arranged based on their values.
- A heap data structure sometimes also called as **BINARY HEAP**.

There are two types of heap data structures.

➤ Max Heap

➤ Min Heap

Every heap data structure has the won properties...

Property #1 (Ordering): Nodes must be arranged in an order according to their values based on Max heap or Min heap.

Property #2 (Structural): All levels in a heap must be full except the last level and all nodes must be filled from left to right strictly.

MAX HEEP

TREE

MAX HEAP

Max heap data structure is a specialized full binary tree data structure. In a max heap nodes are arranged based on node value.

Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes.

Parent Has Greater Value Than Child Nodes

Operations on Max Heap

The following operations are performed on a Max heap data structure...

1.Finding Maximum

2.Insertion

3.Deletion

Finding Maximum Value Operation in Max Heap

- ✓ Finding the node which has maximum value in a max heap is very simple.
- ✓ In a max heap, the root node has the maximum value than all other nodes.
- ✓ So, directly we can display root node value as the maximum value in max heap.

Insertion Operation in Max Heap

Step 1 - Insert the new Node as last leaf from left to right.

Step 2 - Compare new Node value with its Parent node.

Step 3 - If new Node value is greater than its parent, then swap both of them.

Step 4 - Repeat step 2 and step 3 until new Node value is less than its parent node (or) new Node reaches to root.

ALGORITHM

Step 1: [Add the new value and set its POS]

 SET N = N + 1, POS = N

Step 2: SET HEAP[N] = VAL

Step 3: [Find appropriate location of VAL]

 Repeat Steps 4 and 5 while POS > 1

Step 4: SET PAR = POS/2

Step 5: IF HEAP[POS] <= HEAP[PAR],

 then Goto Step 6.

 ELSE

 SWAP HEAP[POS], HEAP[PAR]

 POS = PAR

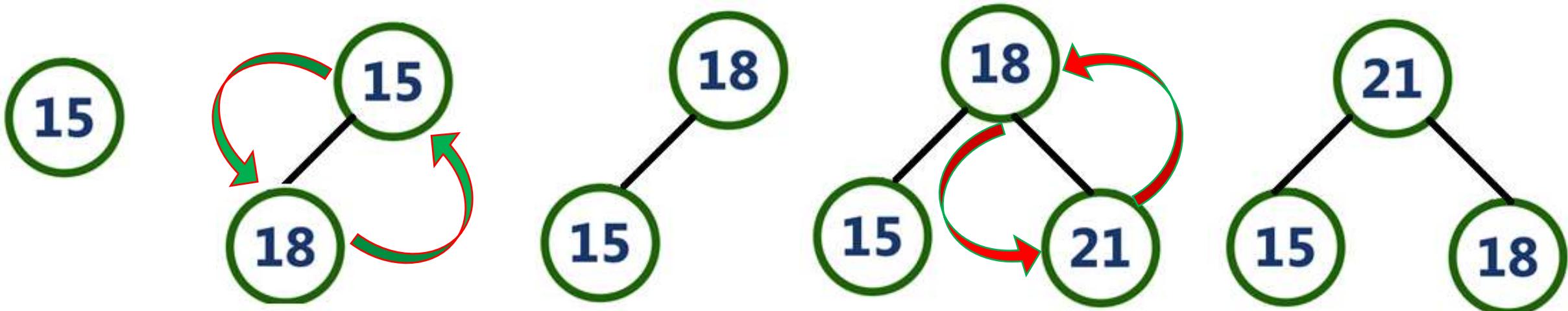
 [END OF IF]

 [END OF LOOP]

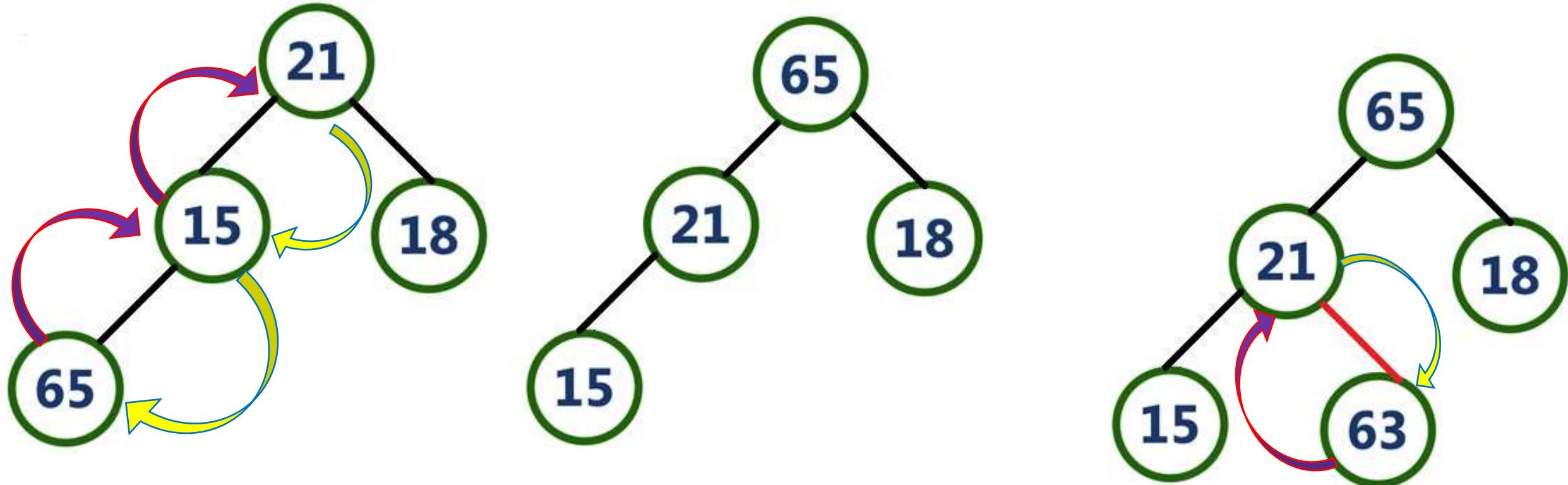
Step 6: RETURN

Example

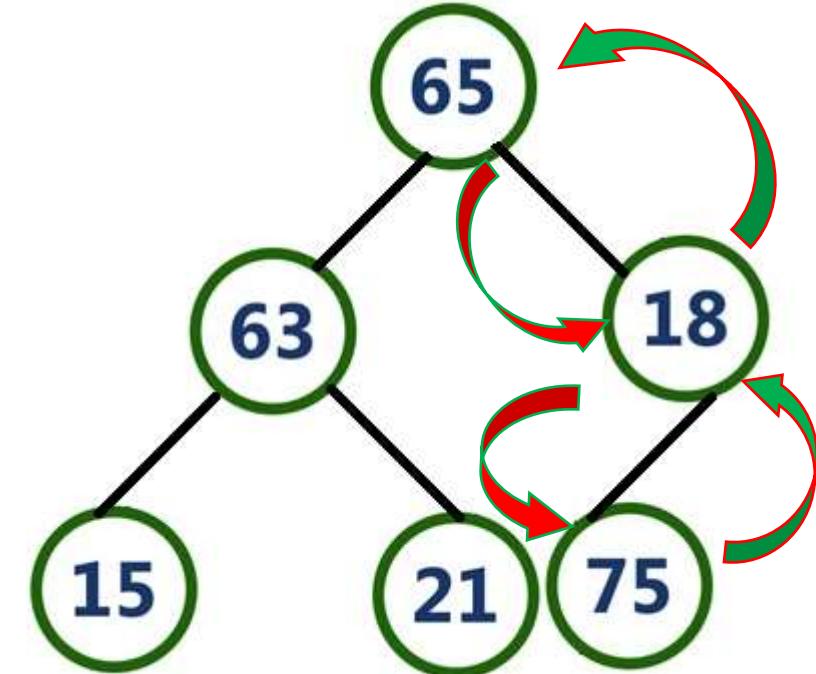
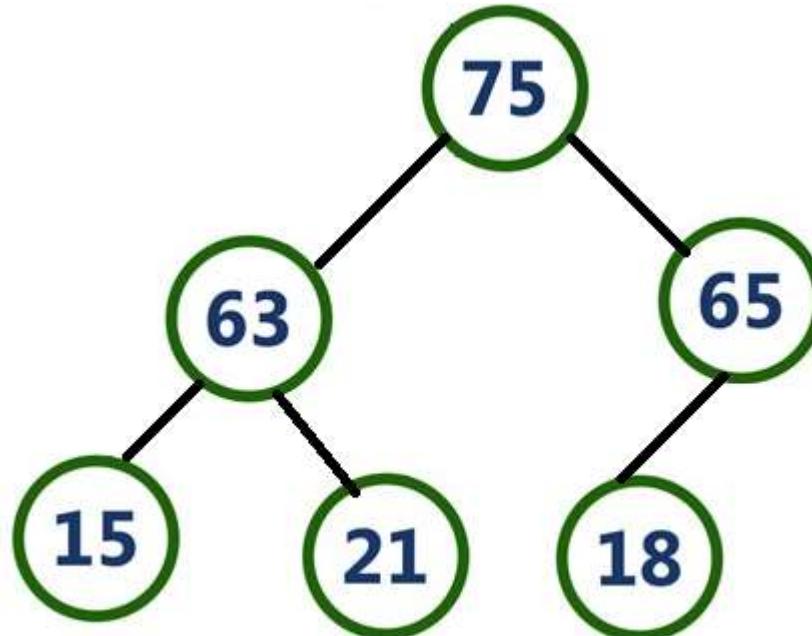
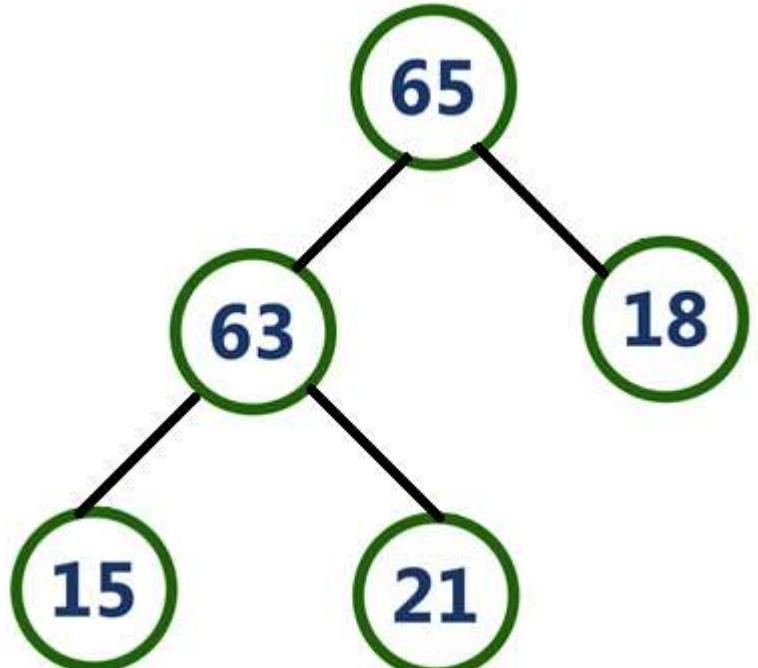
Let us insert the following values 15,18,21,65,63,75,36,70,89,90



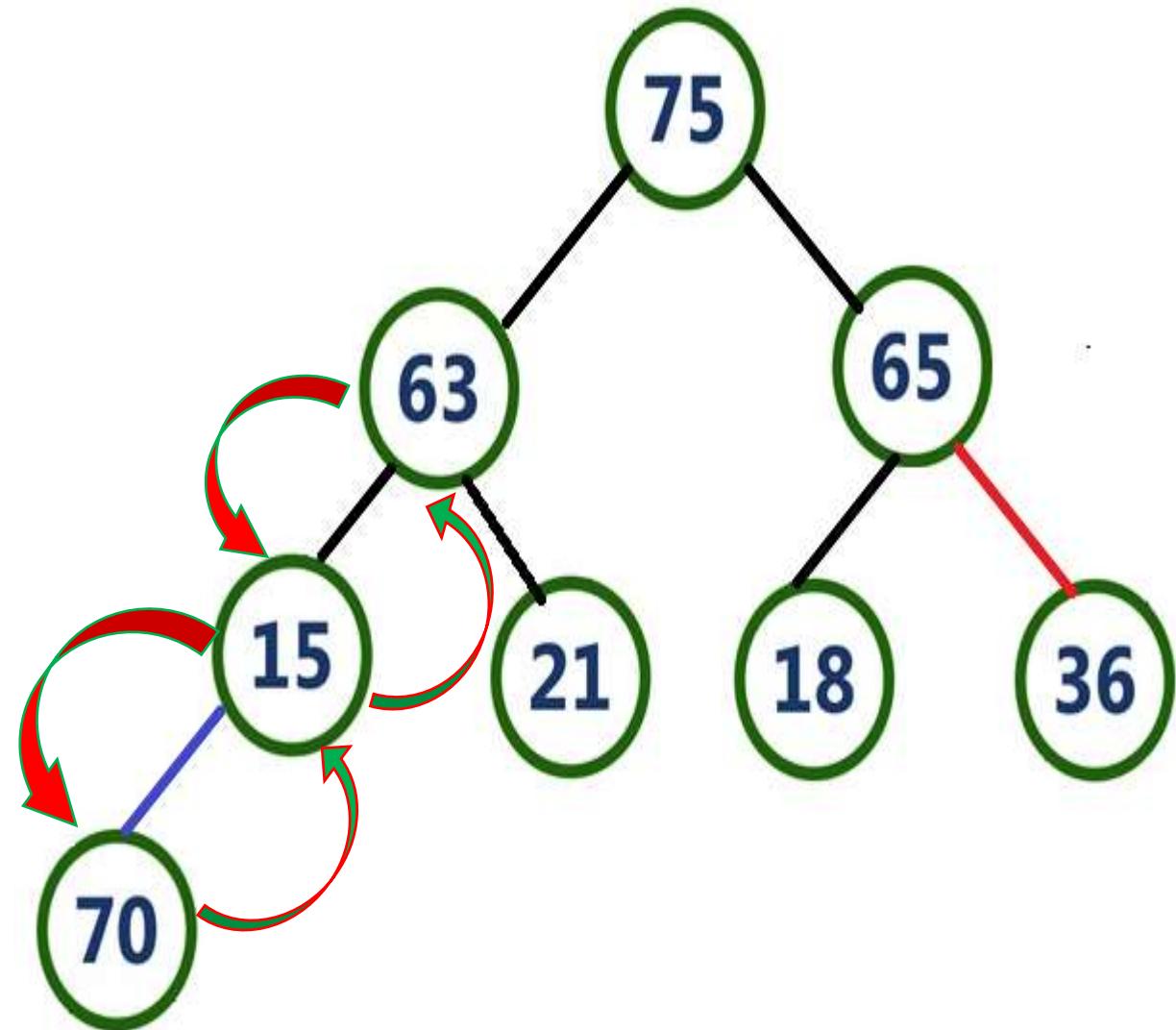
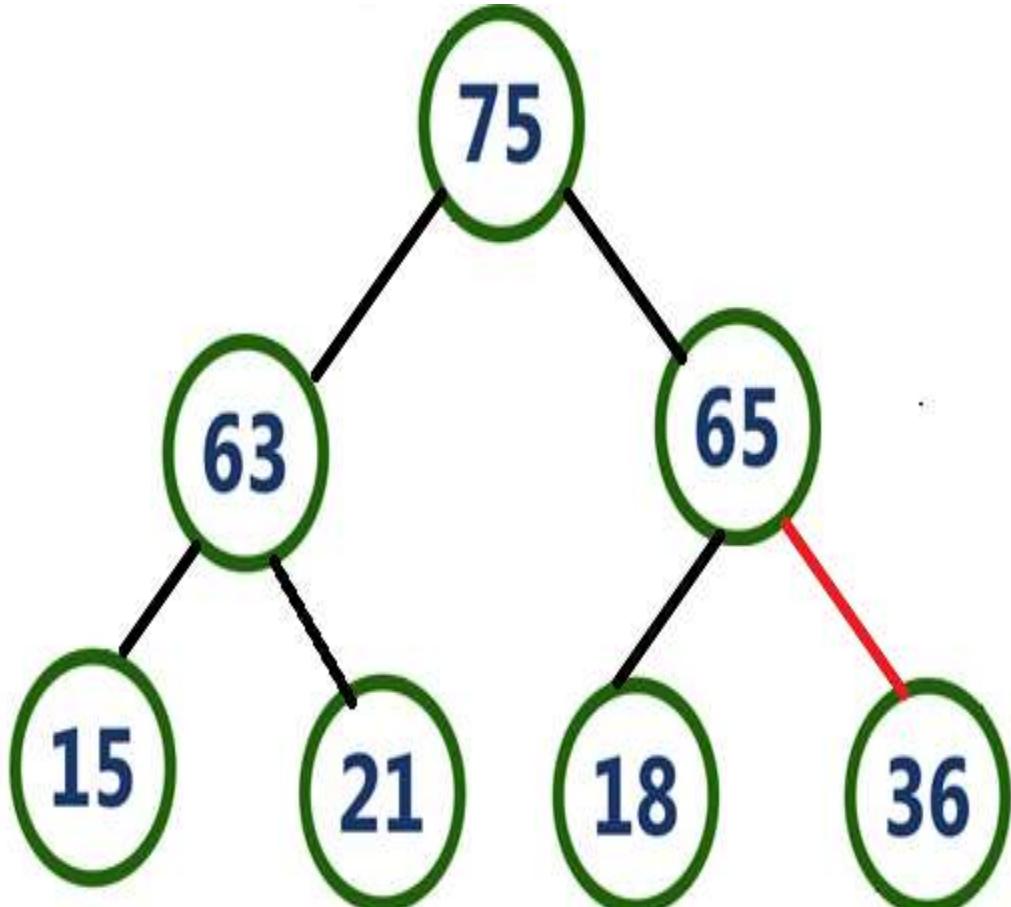
Let us insert the following values 15,18,21,65,63,75,36,70,89,90



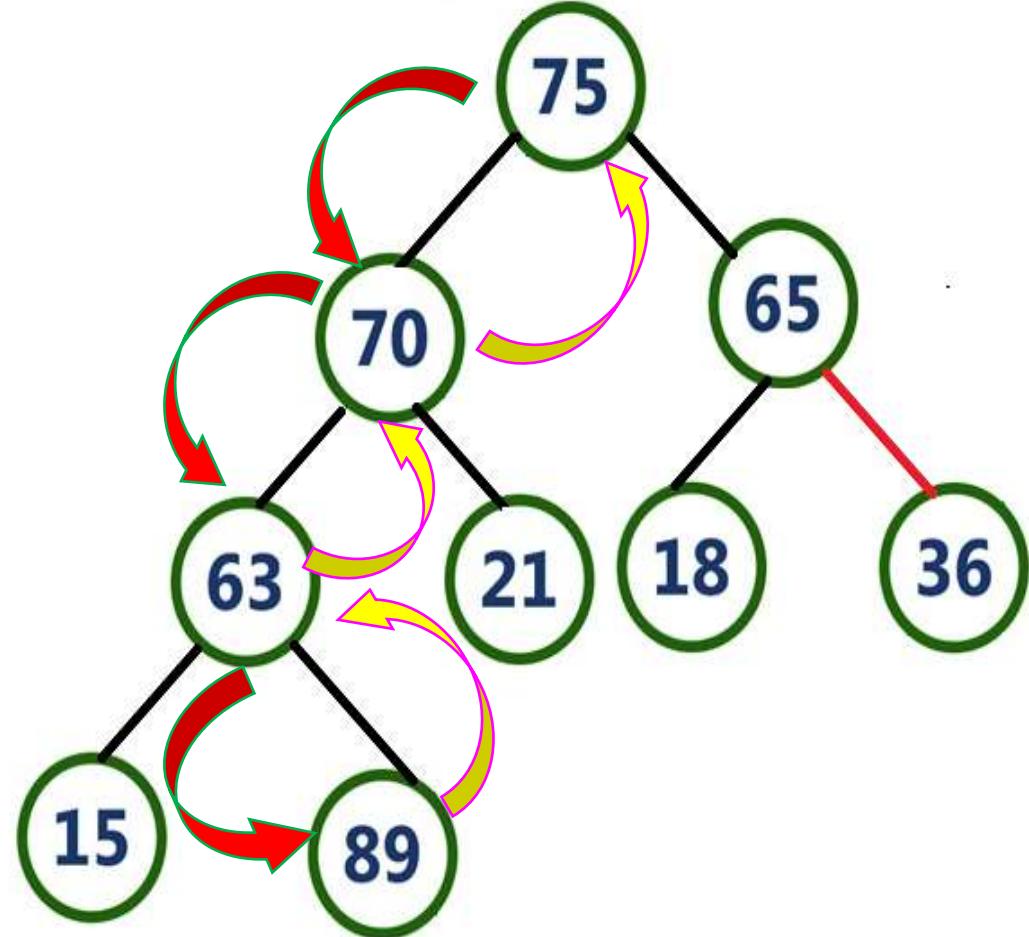
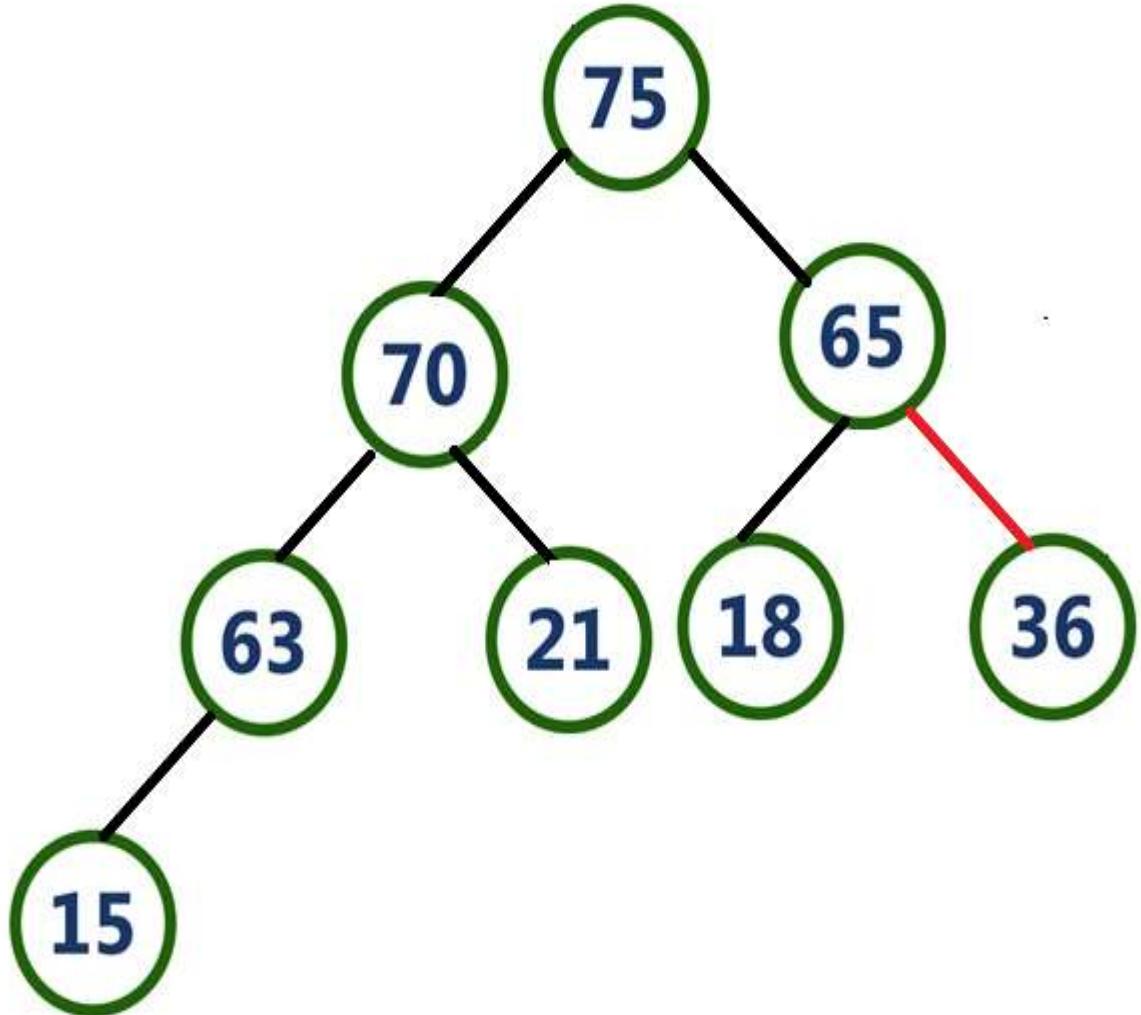
Let us insert the following values 15,18,21,65,63,75,36,70,89,90



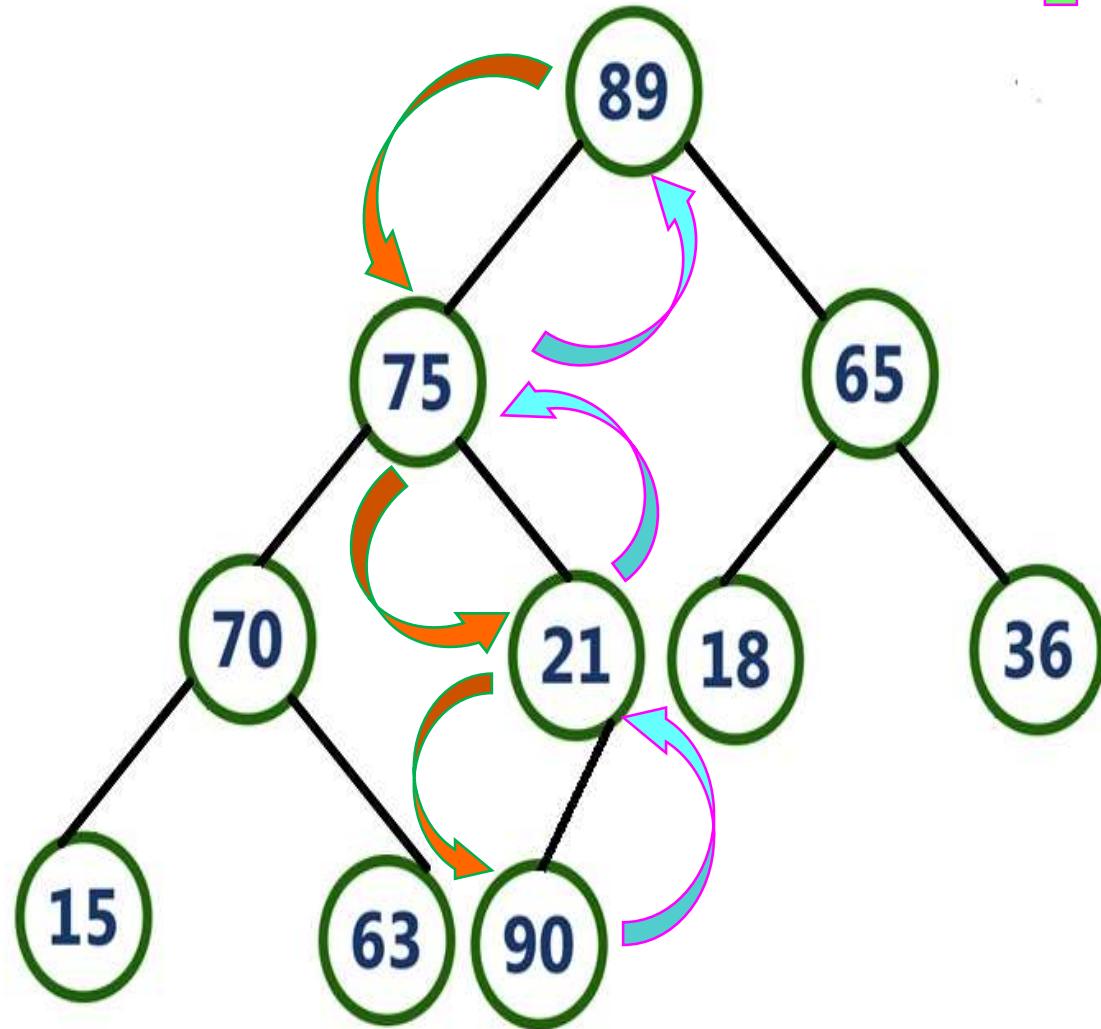
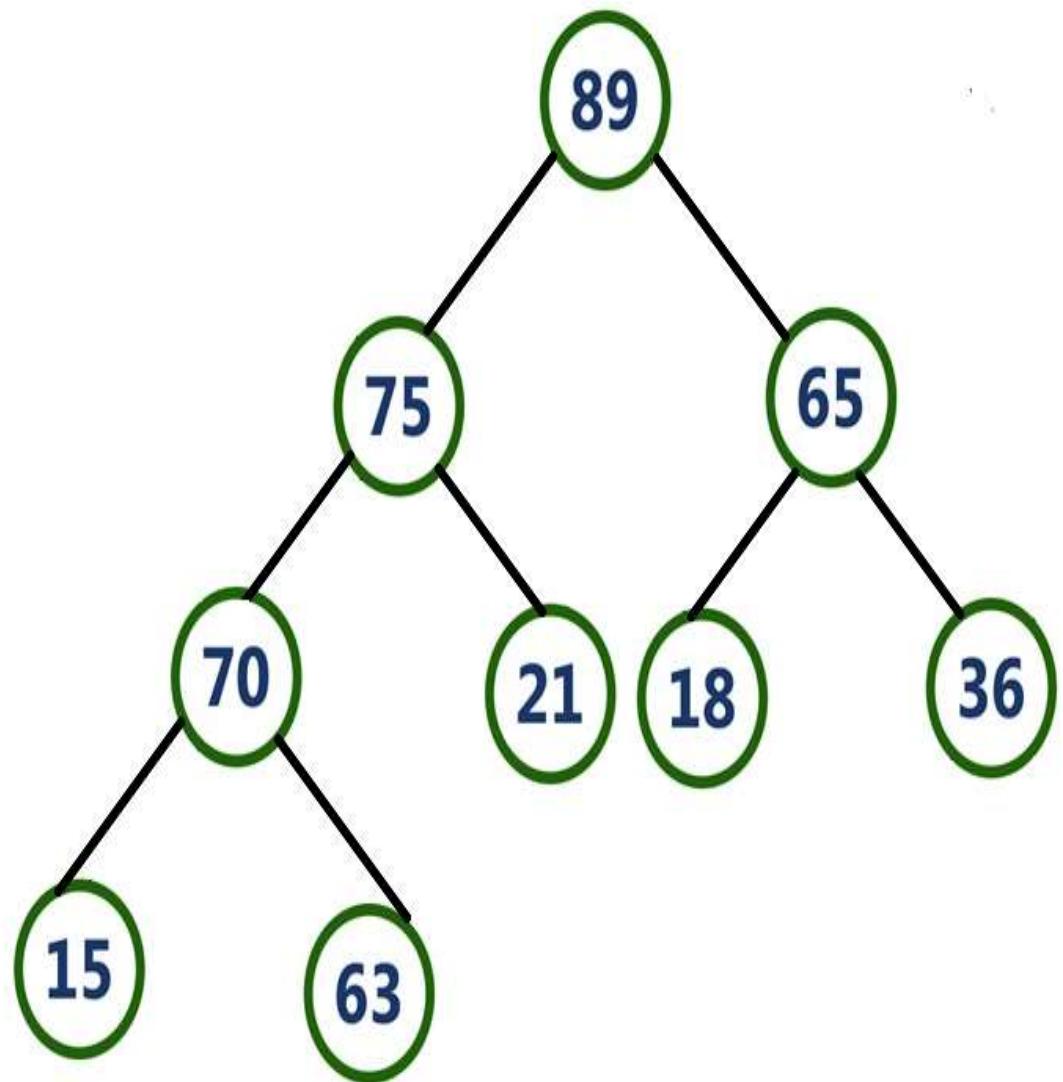
Let us insert the following values 15,18,21,65,63,75,36,70,89,90



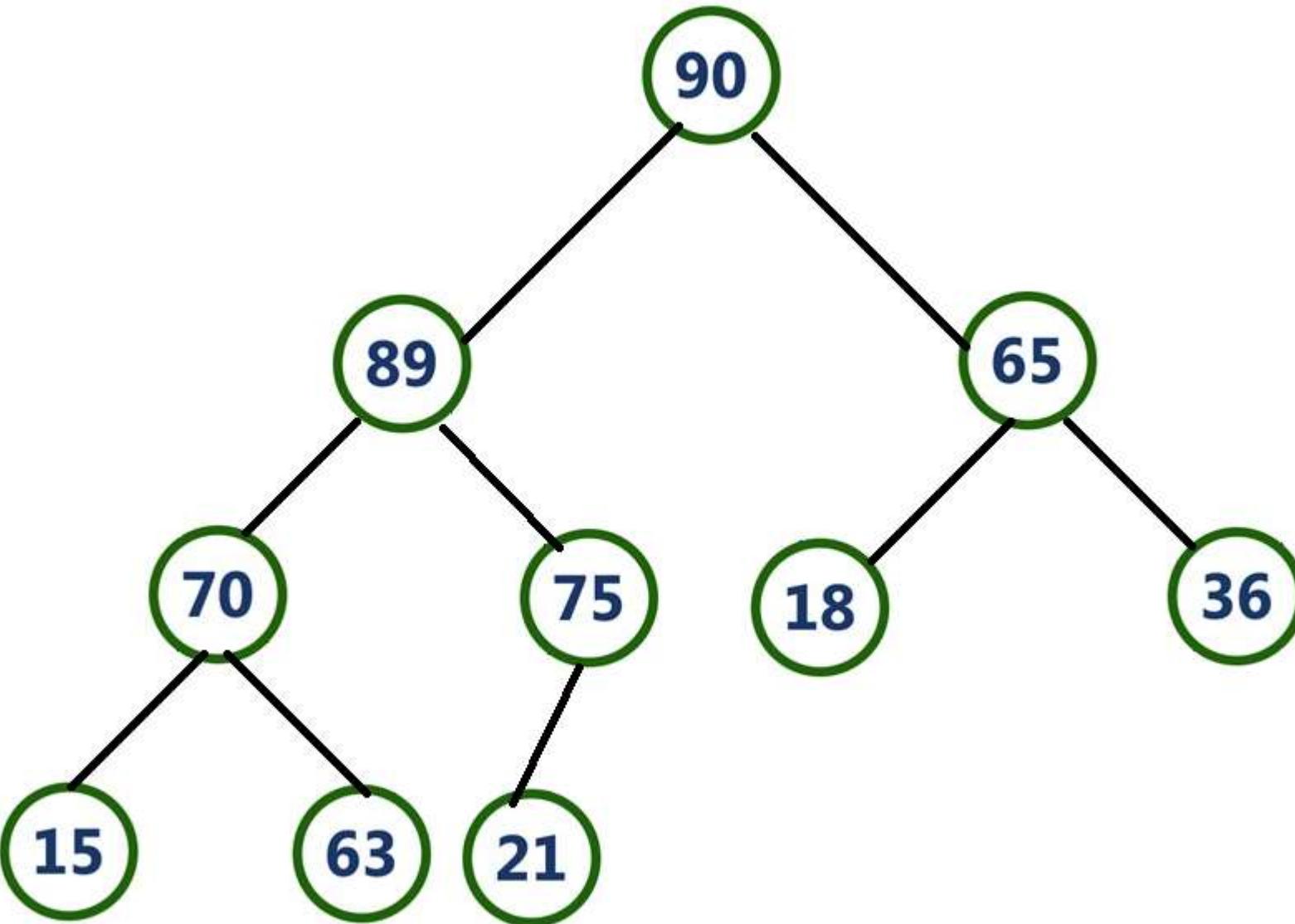
Let us insert the following values 15,18,21,65,63,75,36,70,89,90



Let us insert the following values 15,18,21,65,63,75,36,70,89,90



Let us insert the following values 15,18,21,65,63,75,36,70,89,90



MIN HEEP
TREE

MIN HEAP

Min heap data structure is a specialized full binary tree data structure. In a min heap nodes are arranged based on node value.

Min heap is a specialized full binary tree in which every parent node contains lesser or equal value than its child nodes.

Parent Has lesser Value Than Child Nodes

Operations on Min Heap

The following operations are performed on a Max heap data structure...

1.Finding Minimum

2.Insertion

3.Deletion

Finding Minimum Value Operation in Min Heap

- ✓ Finding the node which has minimum value in a min heap is very simple.
- ✓ In a min heap, the root node has the minimum value than all other nodes.
- ✓ So, directly we can display root node value as the minimum value in min heap.

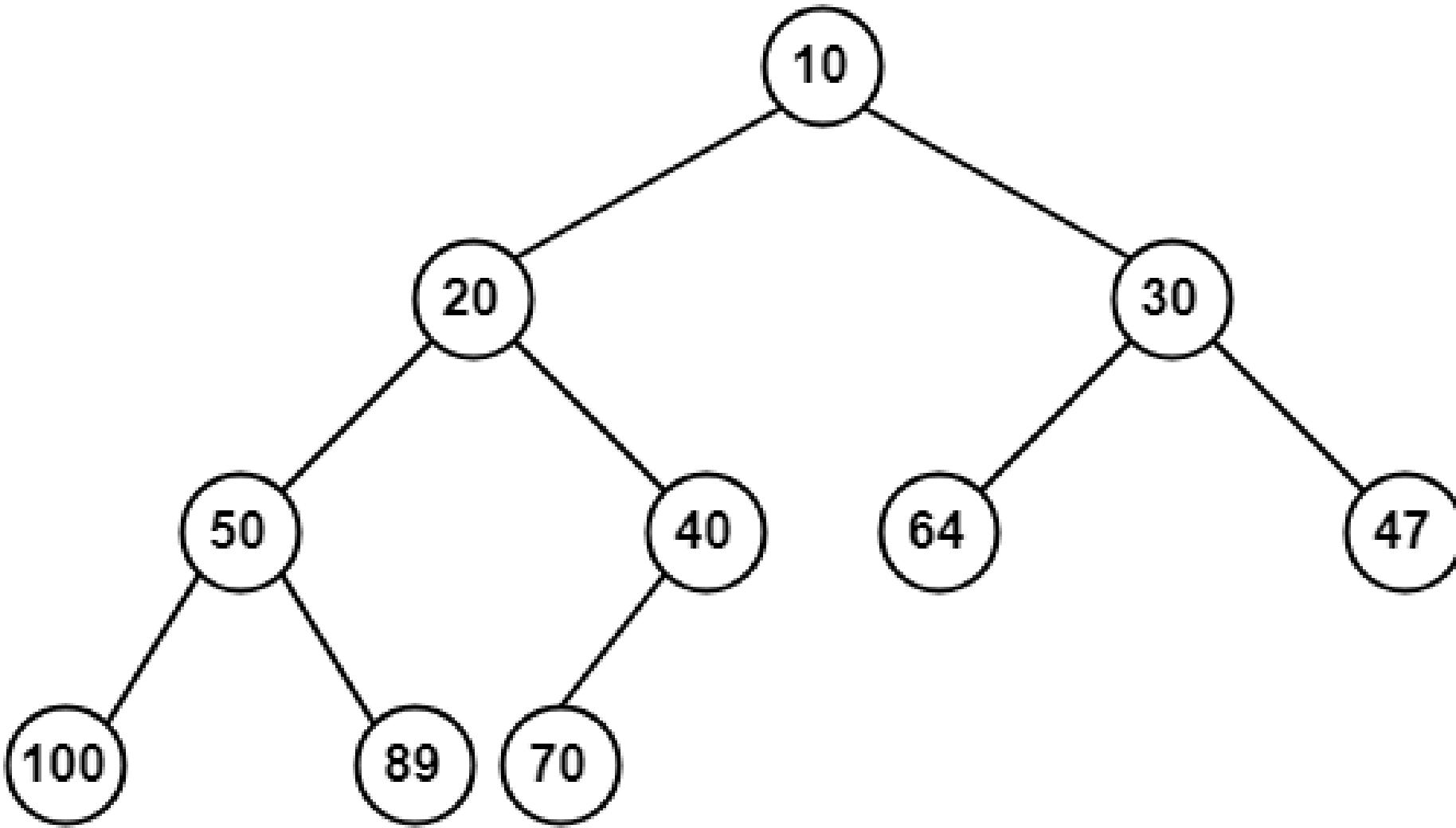
Insertion Operation in Max Heap

Step 1 - Insert the new Node as last leaf from left to right.

Step 2 - Compare new Node value with its Parent node.

Step 3 - If new Node value is lesser than its parent, then swap both of them.

Step 4 - Repeat step 2 and step 3 until new Node value is greater than its parent node (or)
new Node reaches to root.



Min Heap

APPLICATIONS OF HEAPS

Heaps are preferred for applications that includes.

- **Heap-sort.** It is one of the best sorting methods that has no quadratic worst-case scenarios.
- **Selection algorithms.** These algorithms are used to find the minimum, maximum values in linear or sub-linear time.
- **Graph algorithms.** Heaps can be used as internal traversal data structures. This guarantees run time to be reduced by an order of polynomial. Heaps are therefore used for implementing Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.

B+ TREES

What is a B+ Tree?

- A B+ Tree is primarily utilized for implementing dynamic indexing on multiple levels.
- Compared to B- Tree, the B+ Tree stores the data pointers only at the leaf nodes of the Tree, which makes search more process more accurate and faster.

Why use B+ Tree

- Key are primarily utilized to aid the search by directing to the proper Leaf.
- B+ Tree uses a "fill factor" to manage the increase and decrease in a tree.
- In B+ trees, numerous keys can easily be placed on the page of memory because they do not have the data associated with the interior nodes. Therefore, it will quickly access tree data that is on the leaf node.
- A comprehensive full scan of all the elements is a tree that needs just one linear pass because all the leaf nodes of a B+ tree are linked with each other.

Description of B + Trees

- B + Trees is also known as balanced trees
- In which every path from the root of the tree to a leaf of the tree is of the same level.
- B+ Tree index is a multilevel index but it has a structure that differs from that of the multi-level index sequential file

B + Tree Node Structure



- Contains up to n-1 search key values k₁ to k_{n-1}
- Search key values within a node are kept in sorted order i.e i < j than k_i – k_j.
- The data will stored in leaf node only.

Rules for B+ Tree

- Leaves are used to store data records.
- It stored in the internal nodes of the Tree.
- If a target key value is less than the internal node, then the point just to its left side is followed.
- If a target key value is greater than or equal to the internal node, then the point just to its right side is followed.
- The root has a minimum of two children.
- Every leaf node is at same level.
- All the leaf have pointers with each other

For order 4

Maximum child's [m] =4

Maximum keys [m-1] =3

Minimum child's [m/2]-1 =2 =2-1=1

Minimum keys [m/2] =2

Insert the following data

1,4,7,10,17,21,31,25,19,20,28,42

From the order 4 we have 3 key values

1	4	7
---	---	---

INSERT 10

1	4	7	10
---	---	---	----

- After inserting 10 overflow condition occur because max keys is 3
- To overcome this overflow condition split the node

How to split the node

To find the mid element

$$[m/2]+1$$

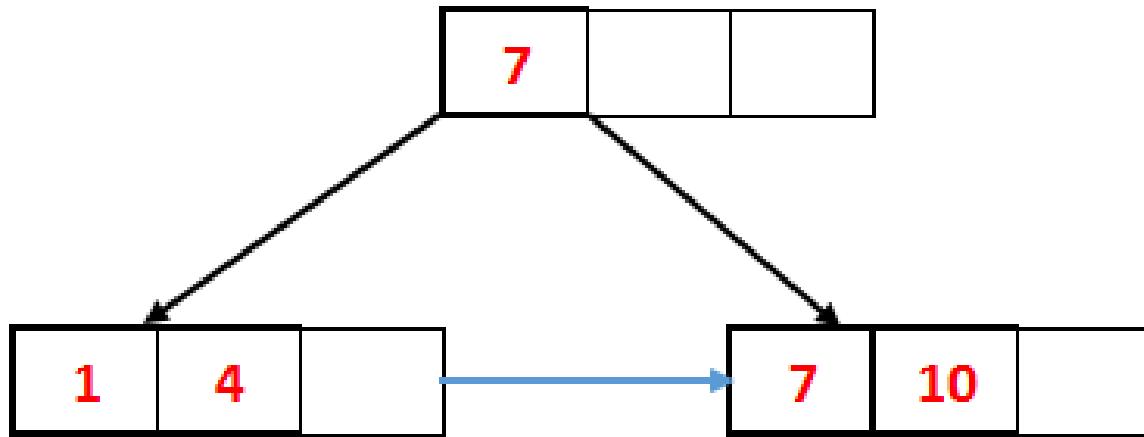
$$=[4/2]+1$$

$$=[2]+1$$

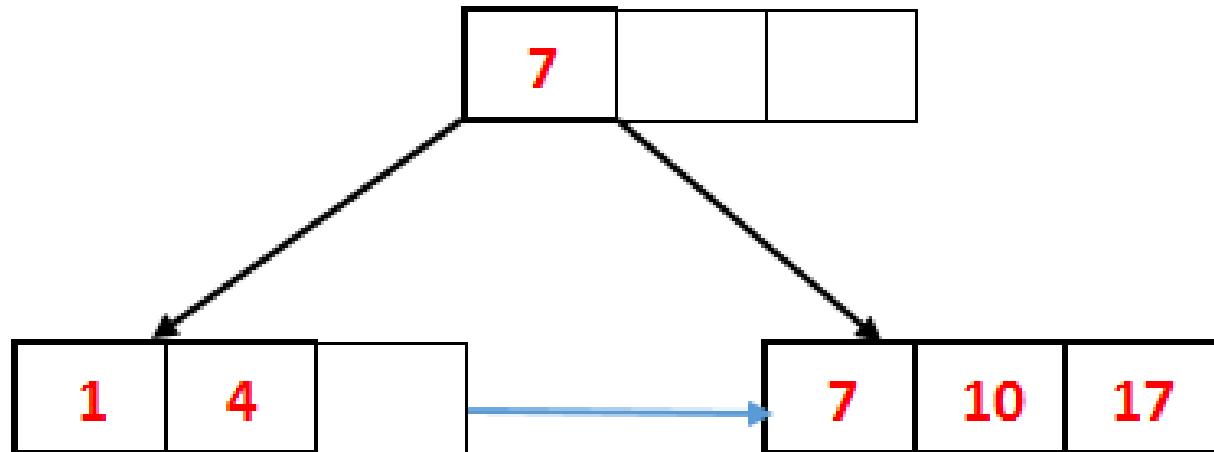
$$=3 \quad =3 \text{ element is the root node}$$

Insert the following data

1,4,7,10,17,21,31,25,19,20,28,42



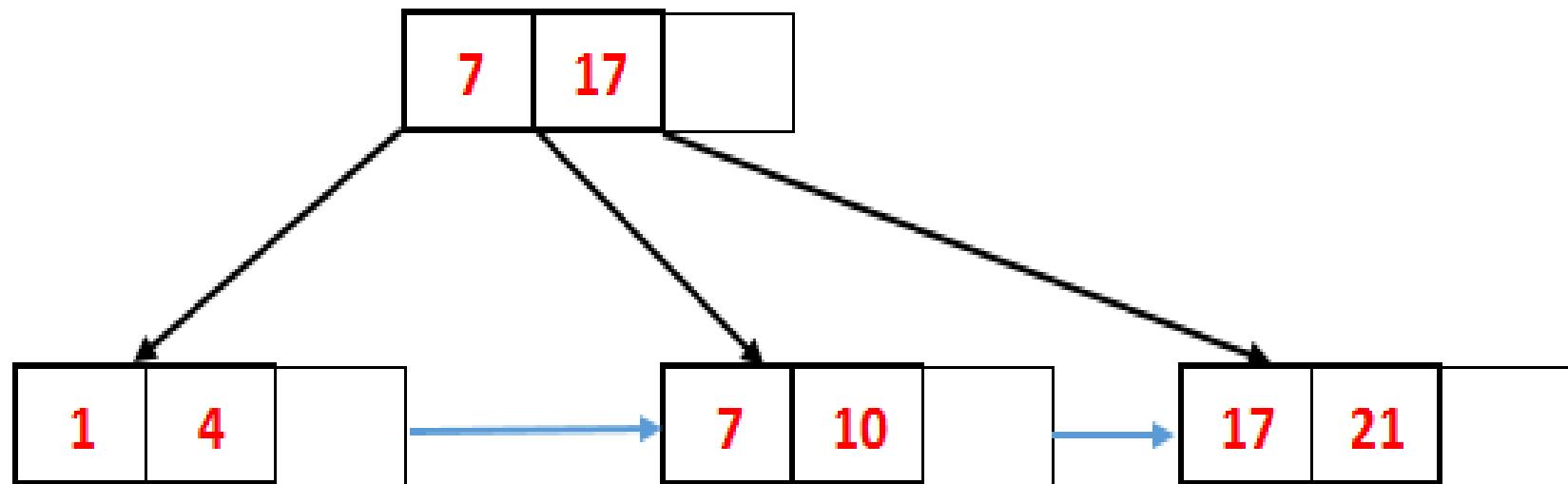
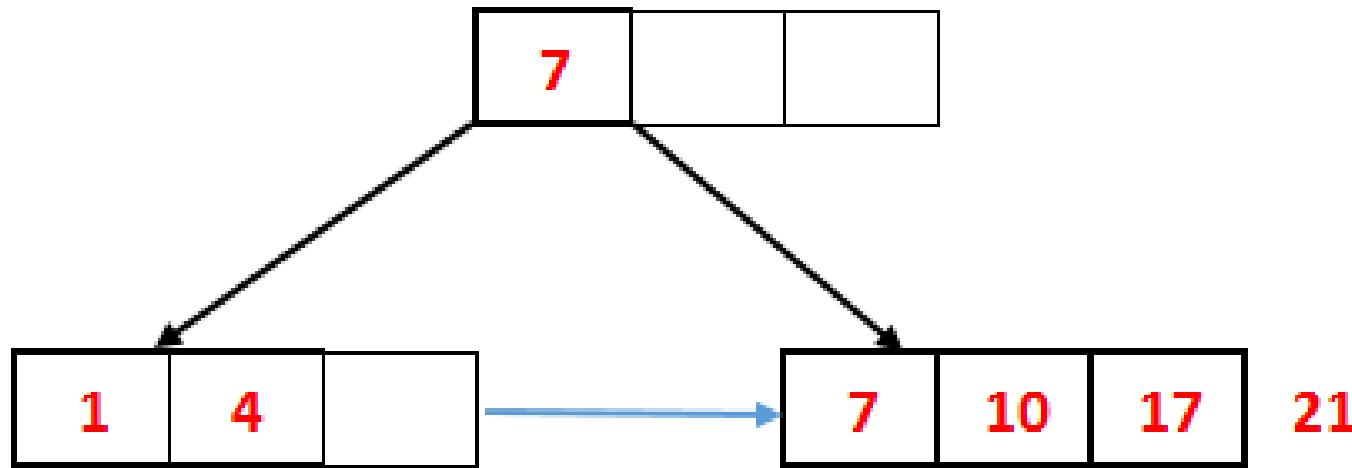
INSERT 17



Insert the following data

1,4,7,10,17,21,31,25,19,20,28,42

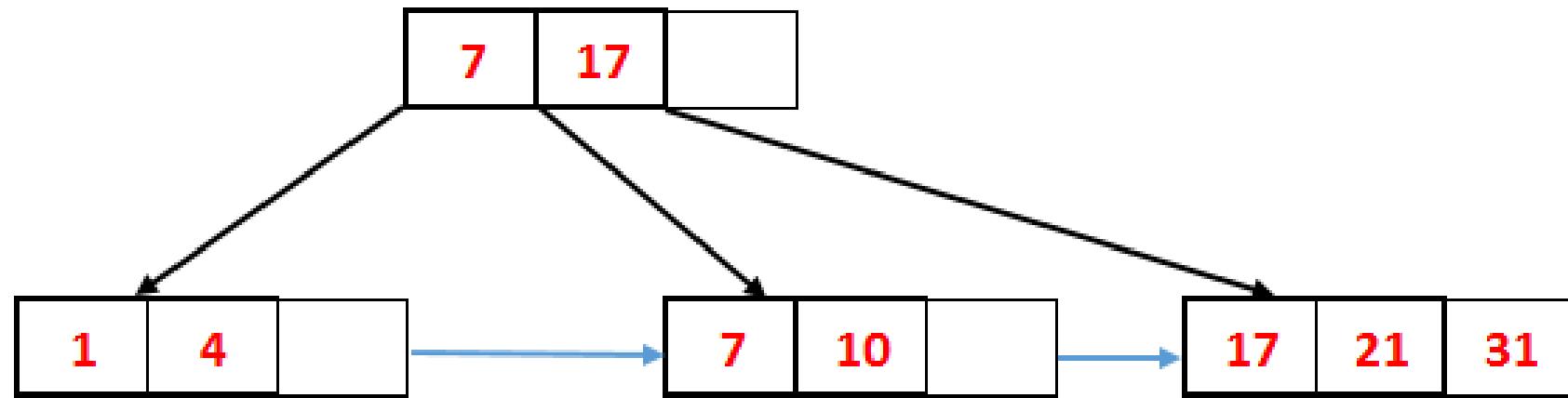
INSERT 21



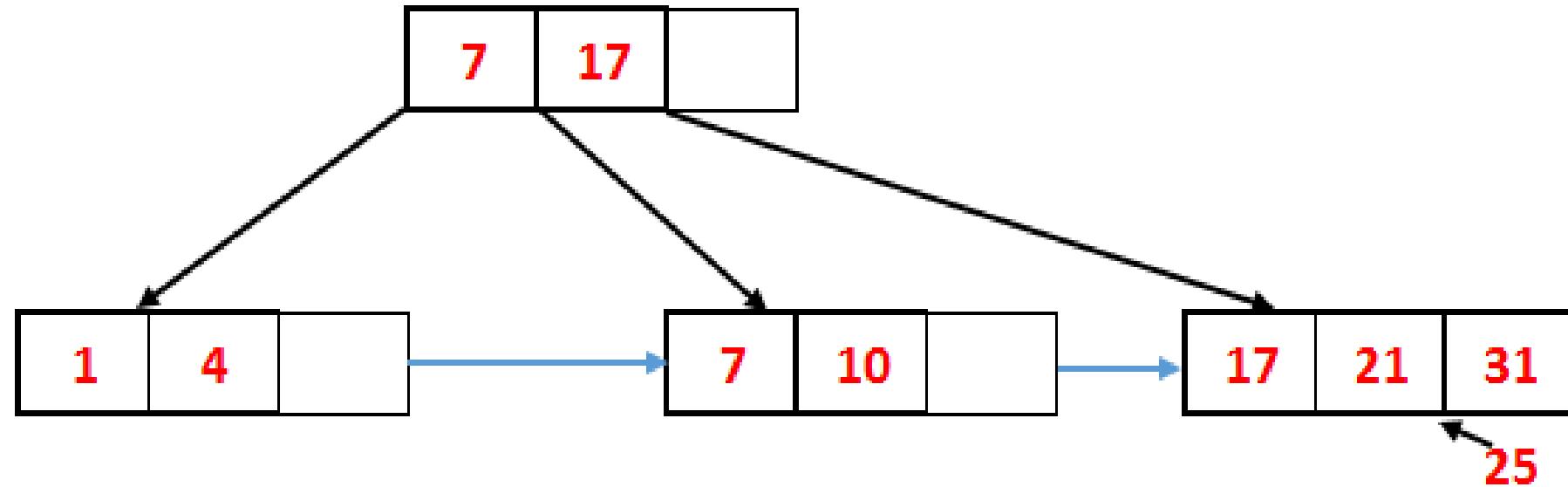
Insert the following data

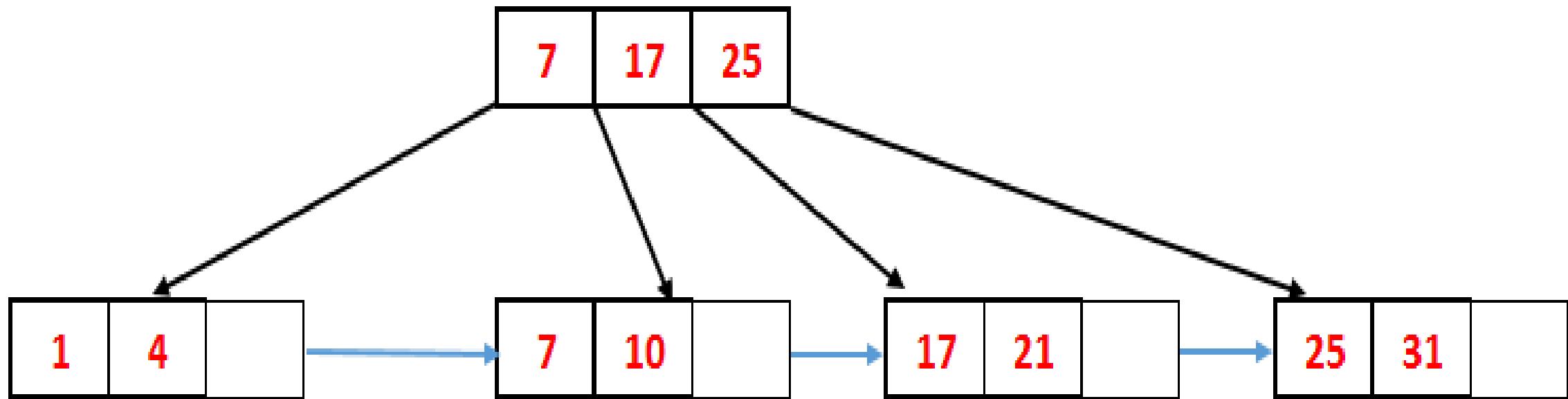
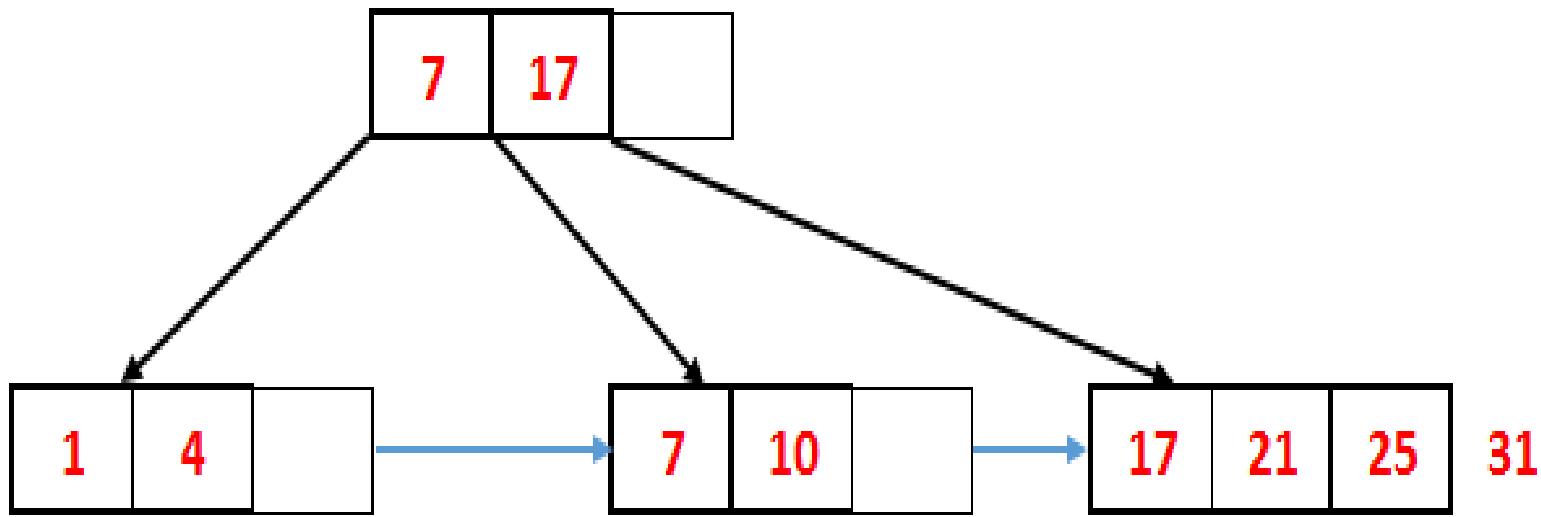
1,4,7,10,17,21,31,25,19,20,28,42

INSERT 31



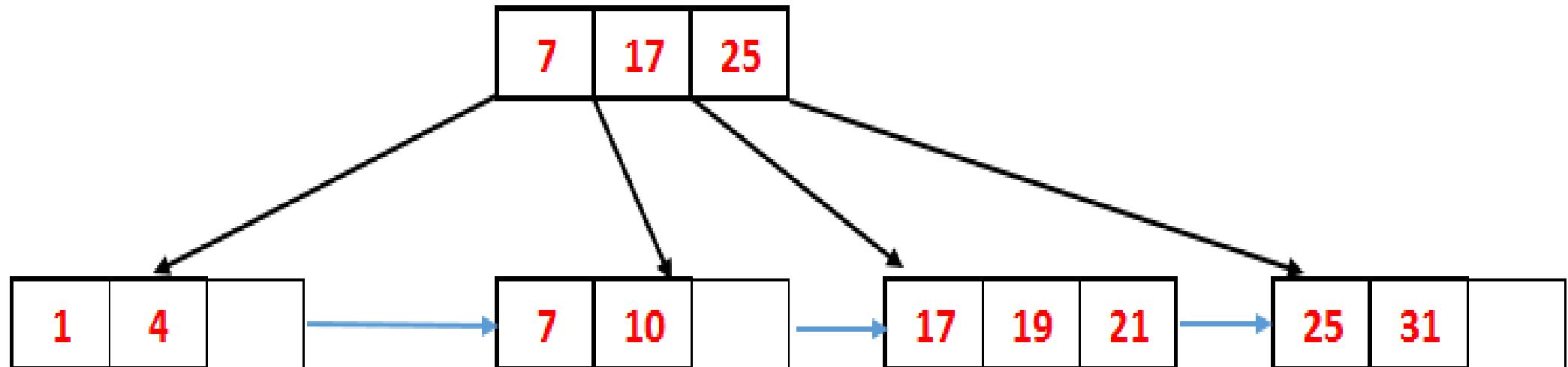
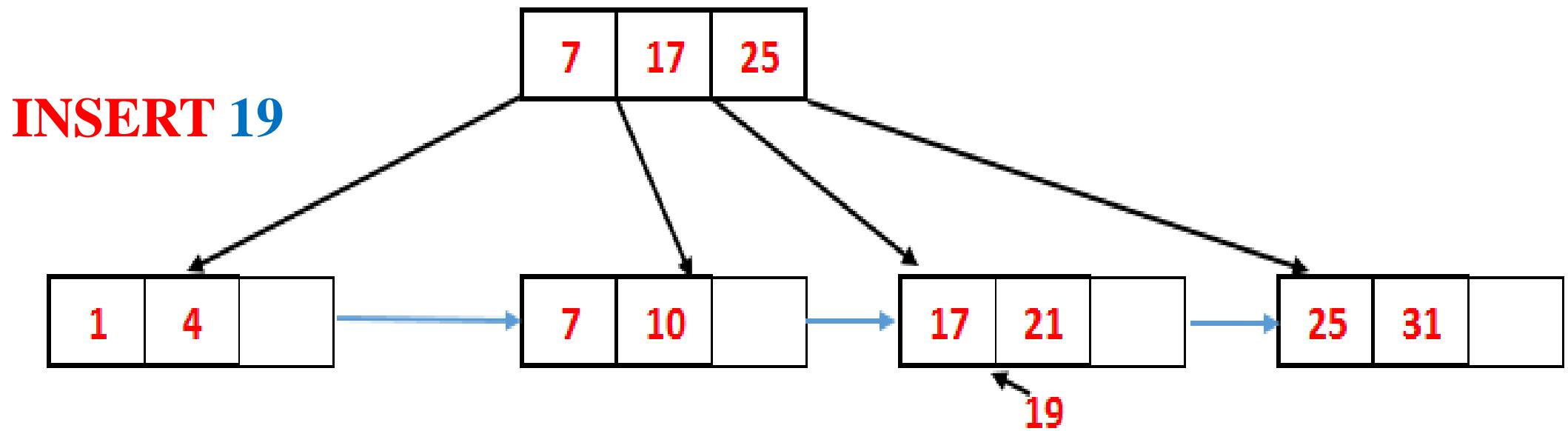
INSERT 25





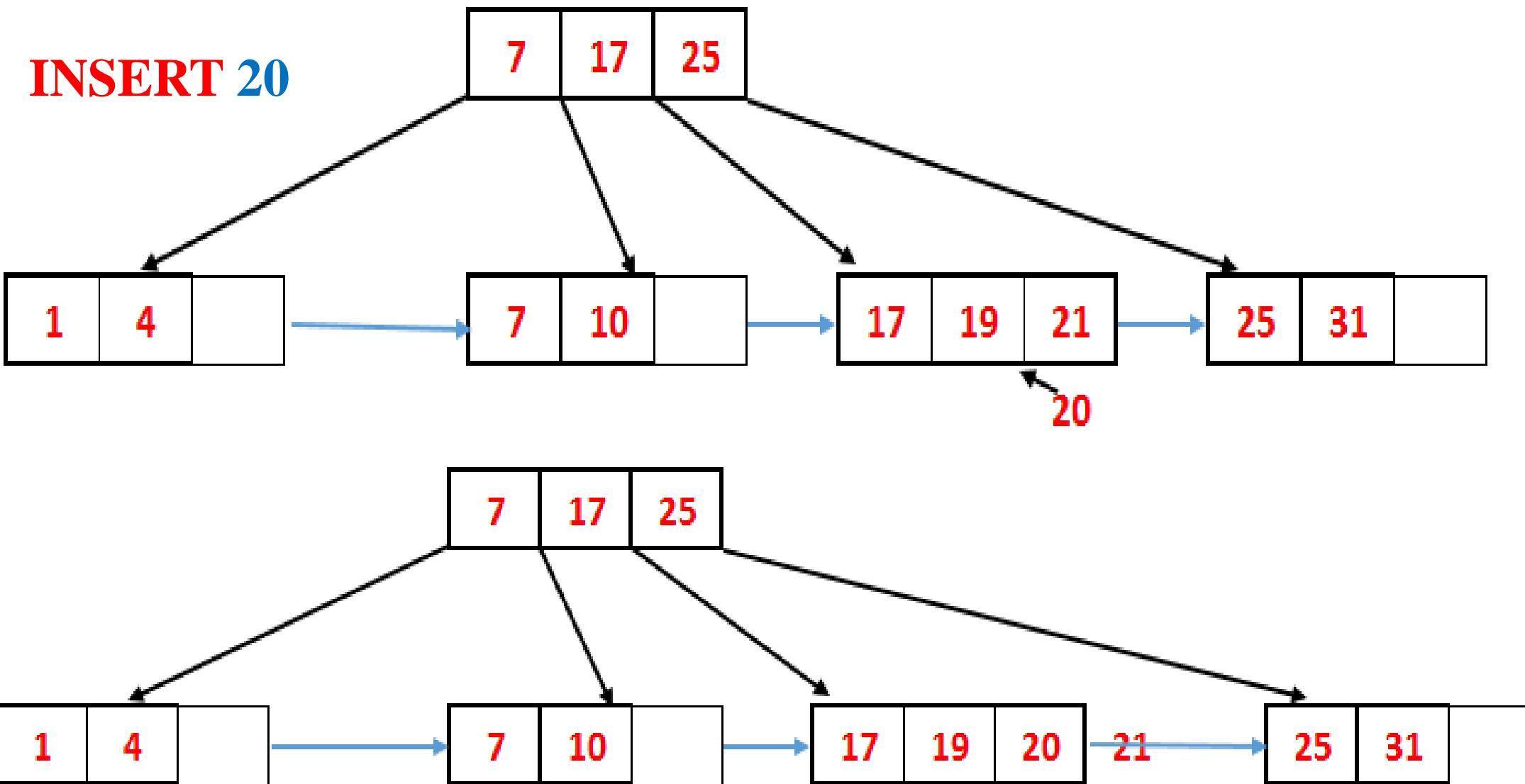
Insert the following data

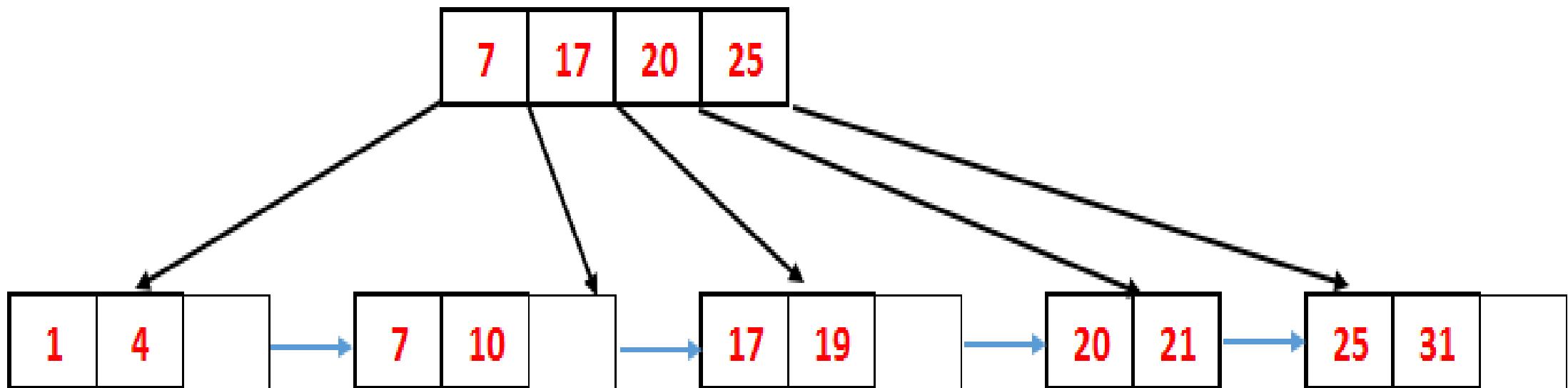
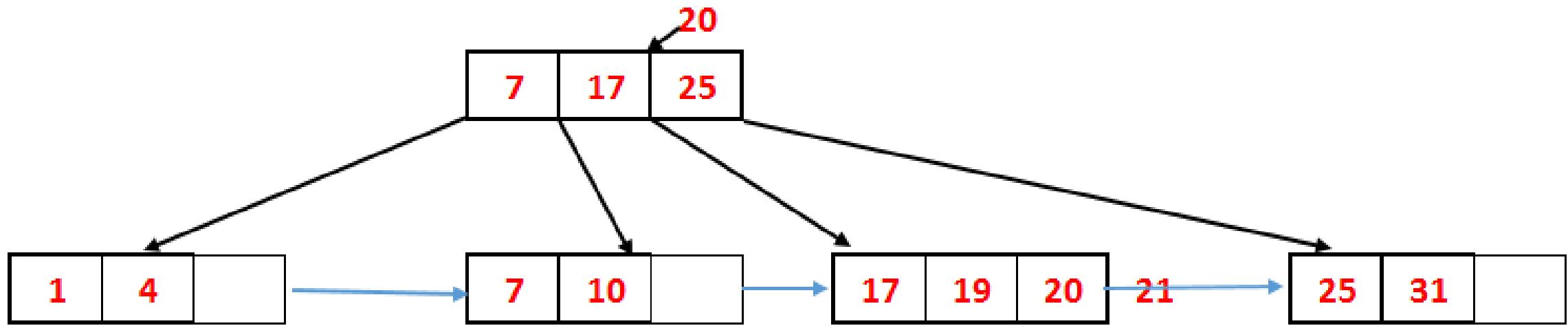
1,4,7,10,17,21,31,25,19,20,28,42

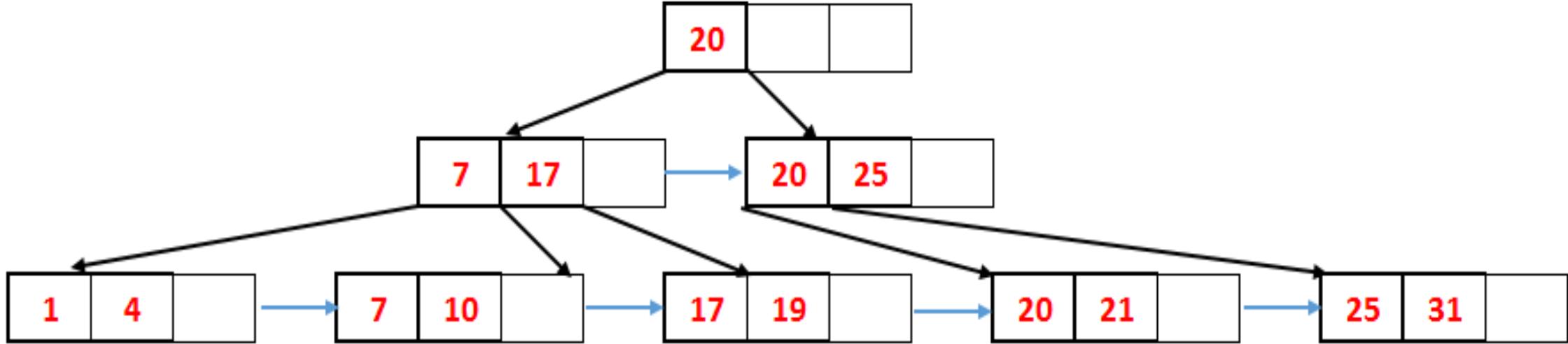


Insert the following data

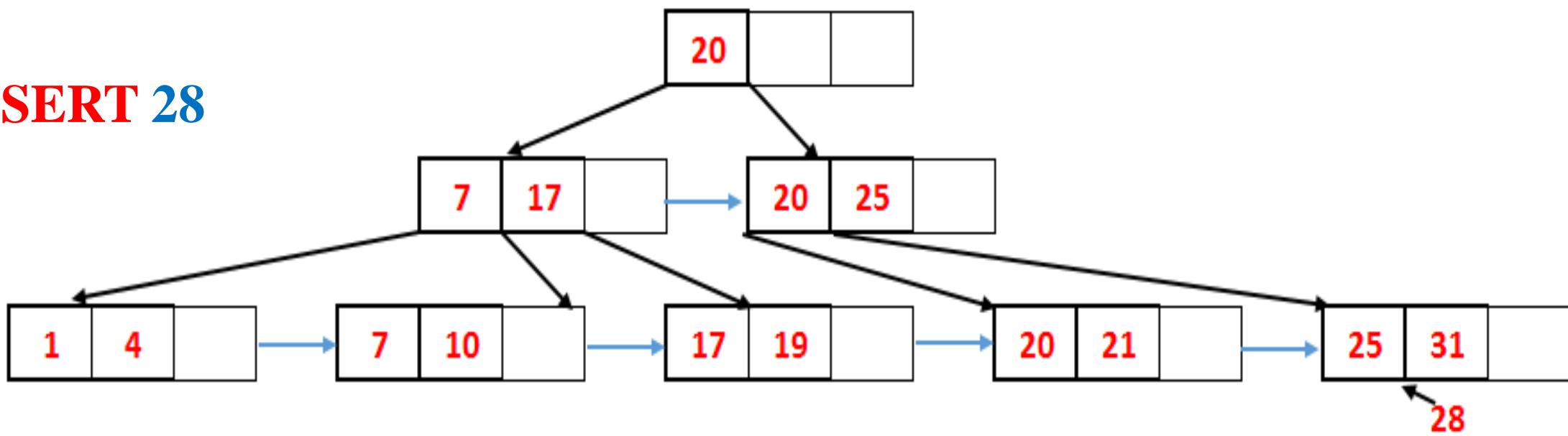
1,4,7,10,17,21,31,25,19,20,28,42

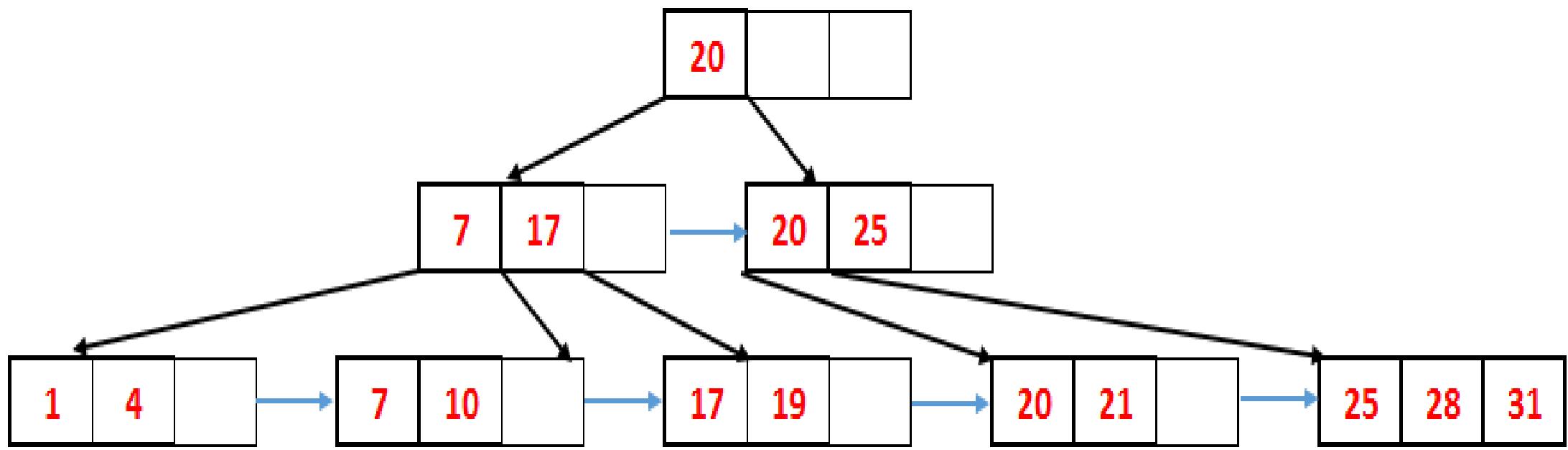




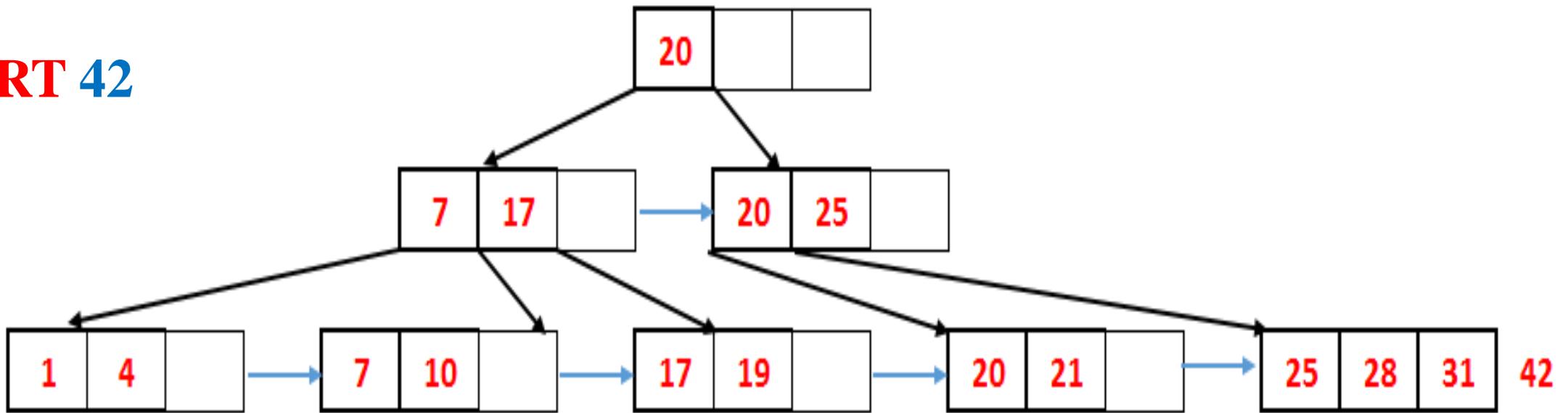


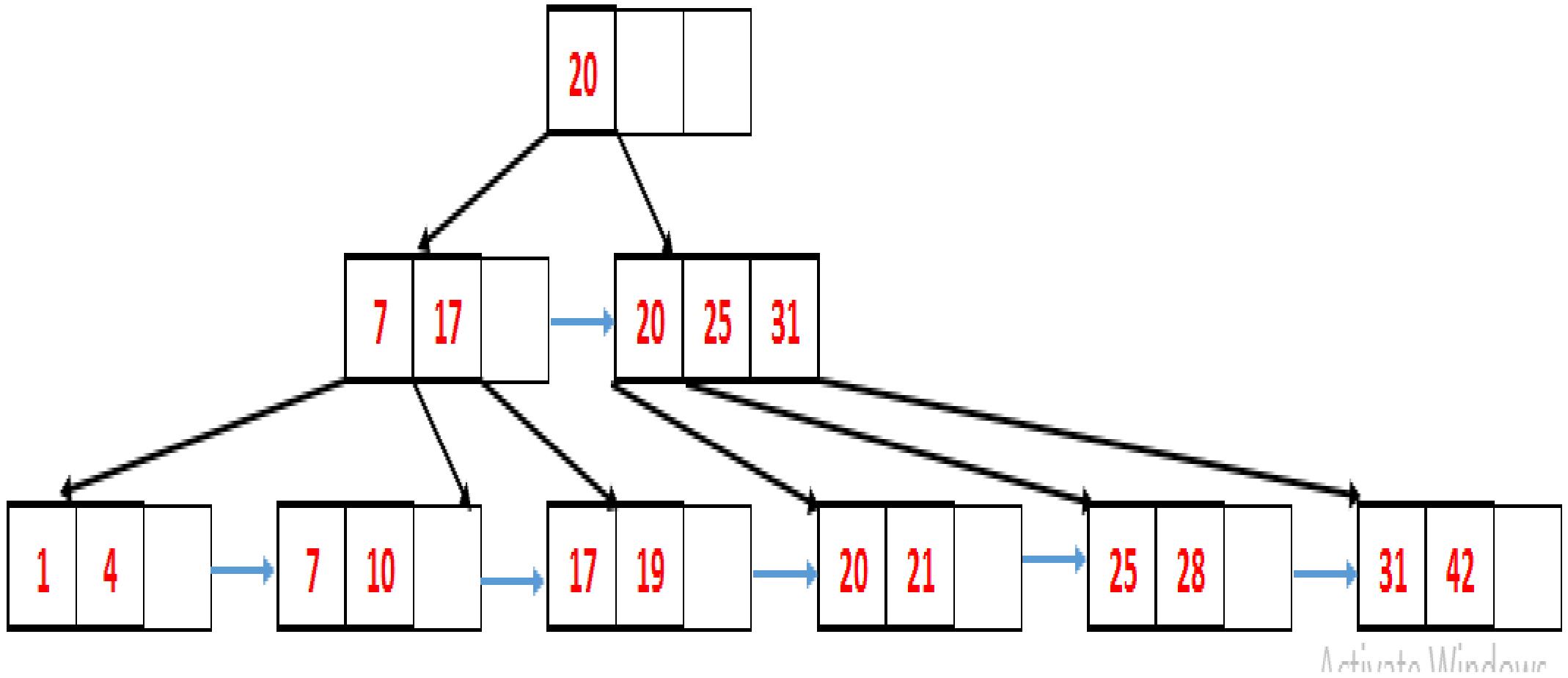
INSERT 28





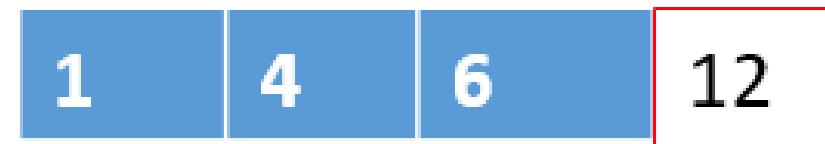
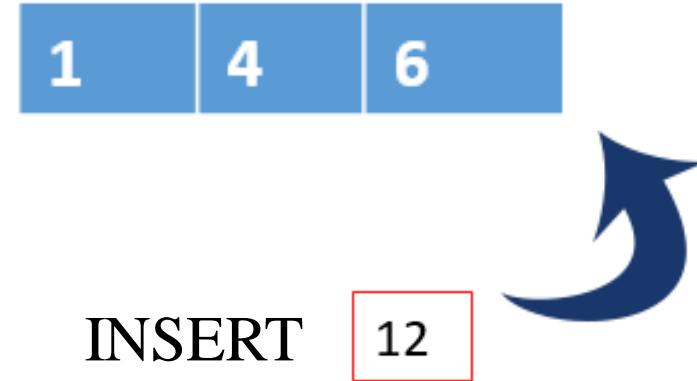
INSERT 42



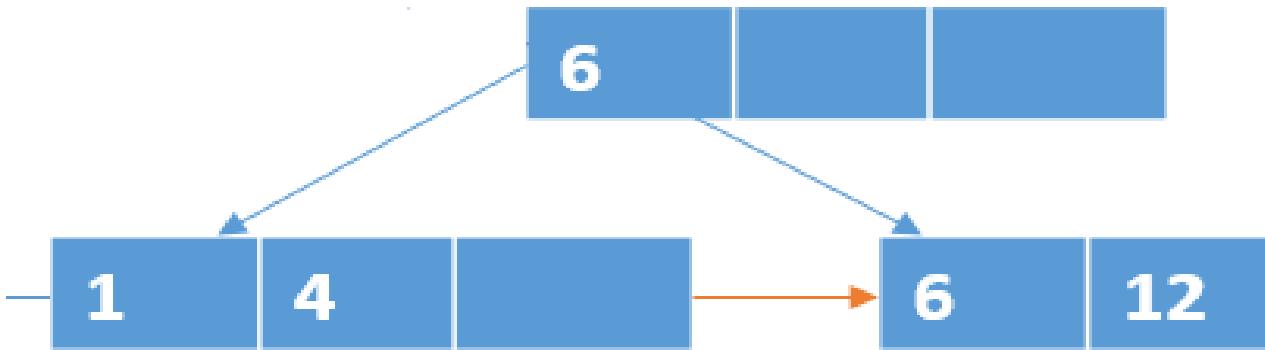


Insert the following data

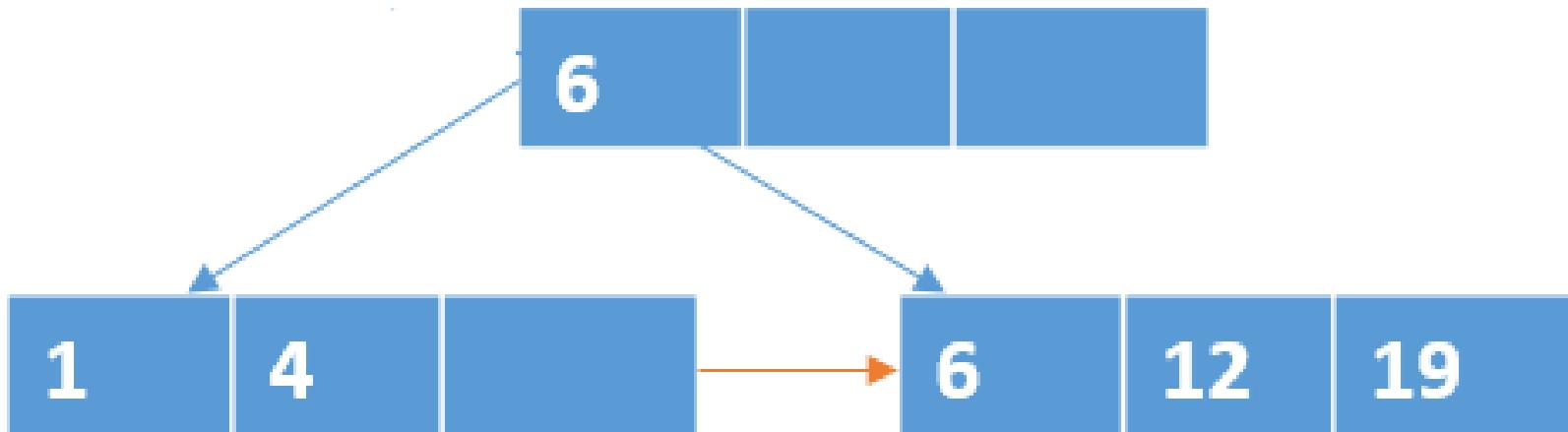
1,4,6,12,19,21,31



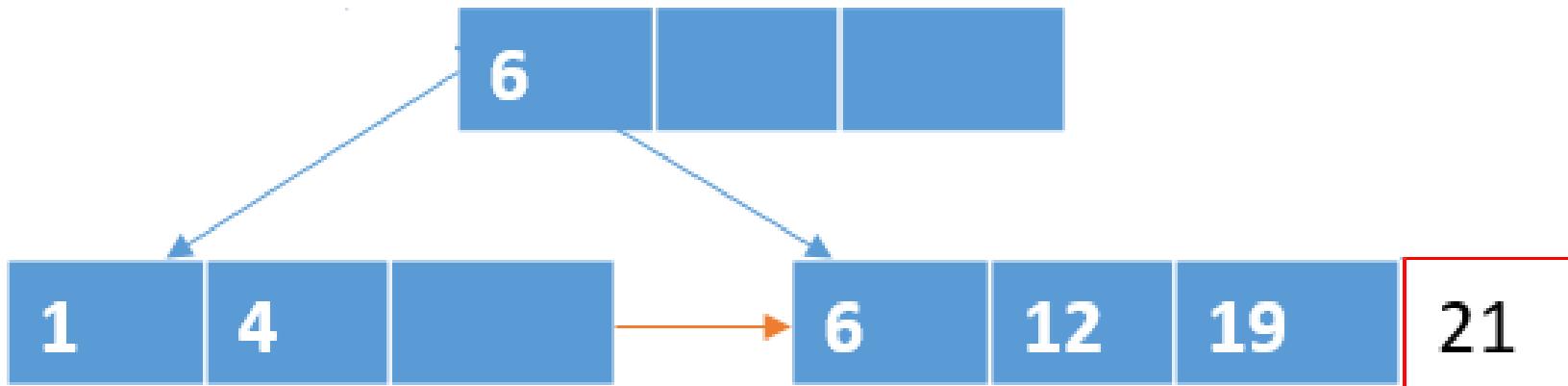
- After inserting 12 overflow condition occur because max keys is **3**
- To overcome this overflow condition split the node



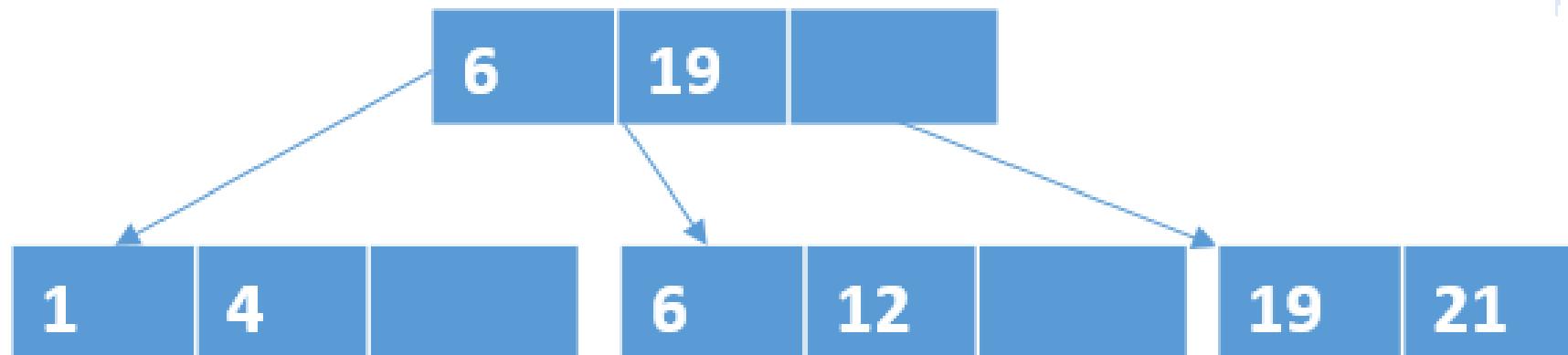
INSERT 19



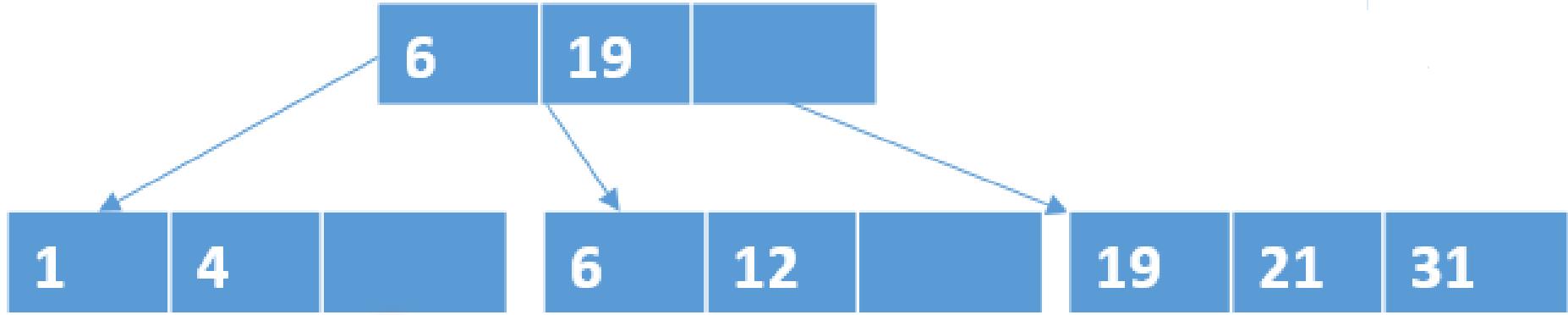
INSERT 21



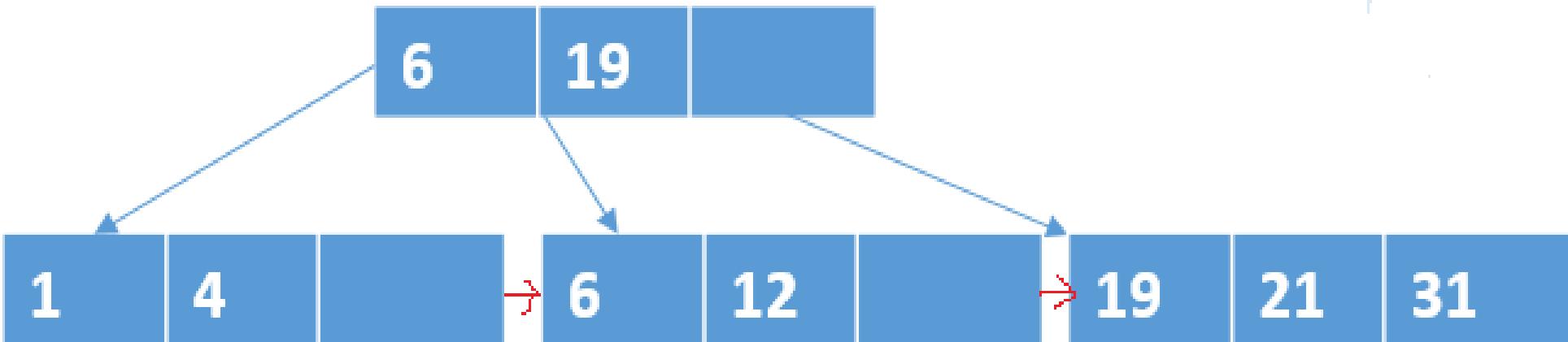
- After inserting 21 overflow condition occur because max keys is 3
- To overcome this overflow condition split the node



INSERT 31



INTERMEDIATE STAGE



DELETING A VALUE FROM A BINARY TREE

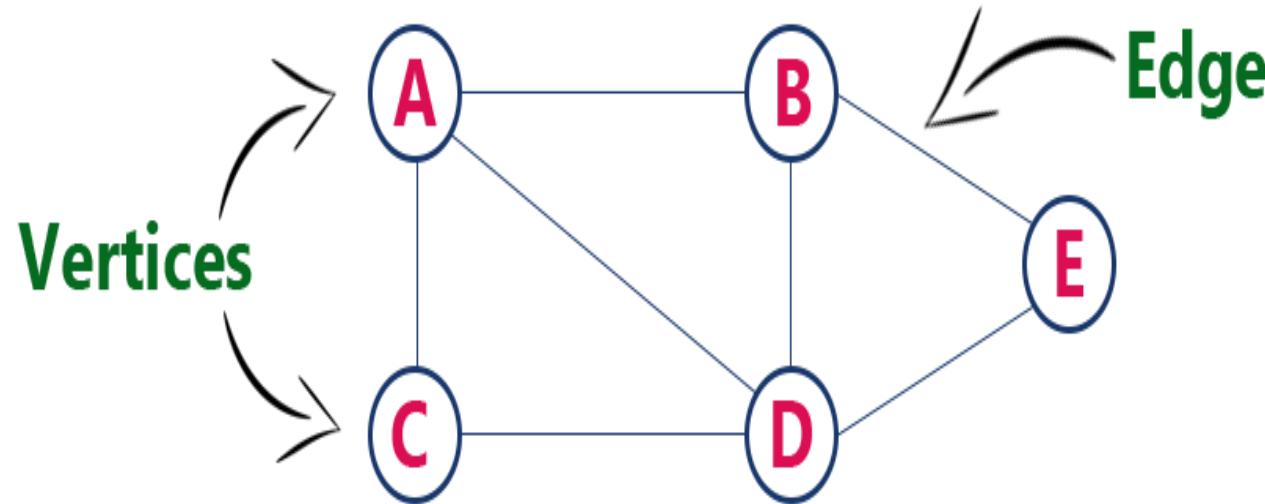
GRAPHS

DEFINITION

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is defined as follows...

- **Graph is a collection of vertices and arcs which connects vertices in the graph**
- **Graph is a collection of nodes and edges which connects nodes in the graph**
- **Generally, a graph G is represented as $G = (V, E)$, where V is set of vertices and E is set of edges.**

GRAPHS



Example

✓ The following is a graph with 5 vertices and 7 edges.

This graph G can be defined as $G = (V, E)$

Where $V = \{A, B, C, D, E\}$ and

$E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}.$

Graph Terminology

Vertex

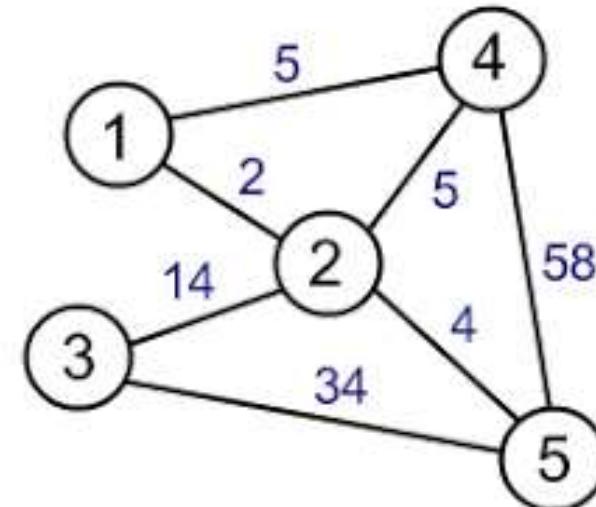
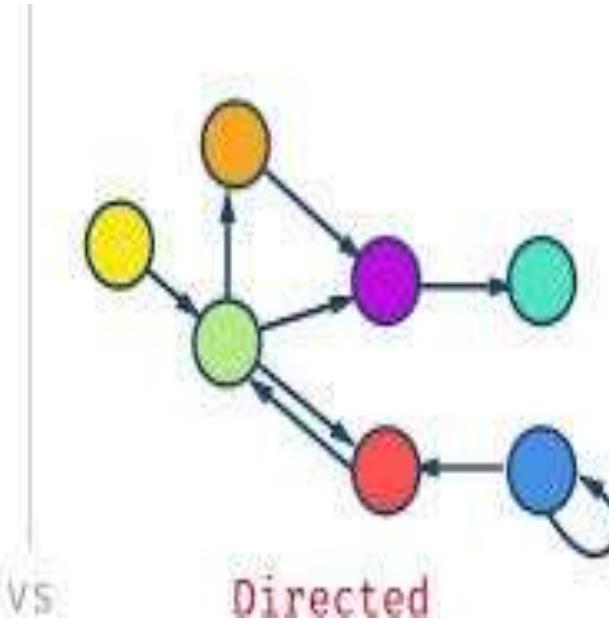
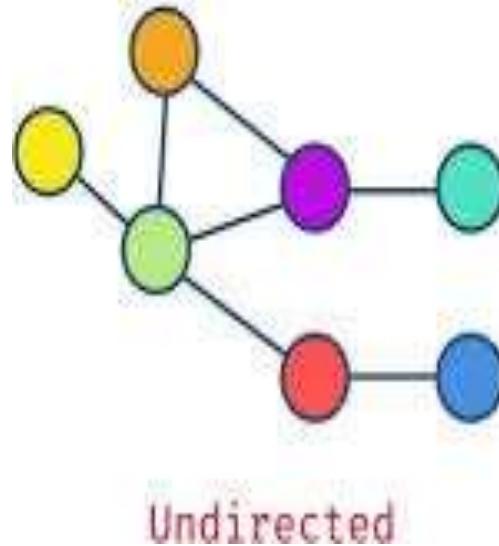
A individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge

- An edge is a connecting link between two vertices.
- **Edge** is also known as **Arc**.
- An edge is represented as (starting Vertex, ending Vertex).
- For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

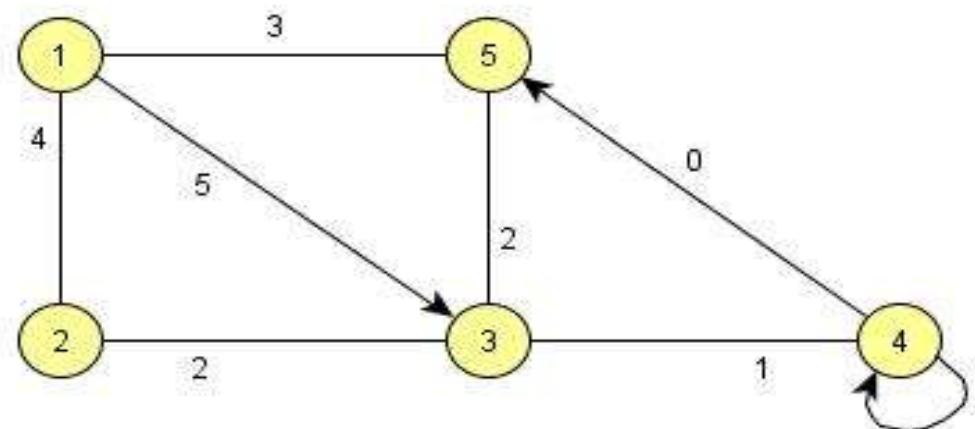
Edges are three types.

- ✓ **Undirected Edge** - An undirected edge is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
- ✓ **Directed Edge** - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
- ✓ **Weighted Edge** - A weighted edge is an edge with cost on it.

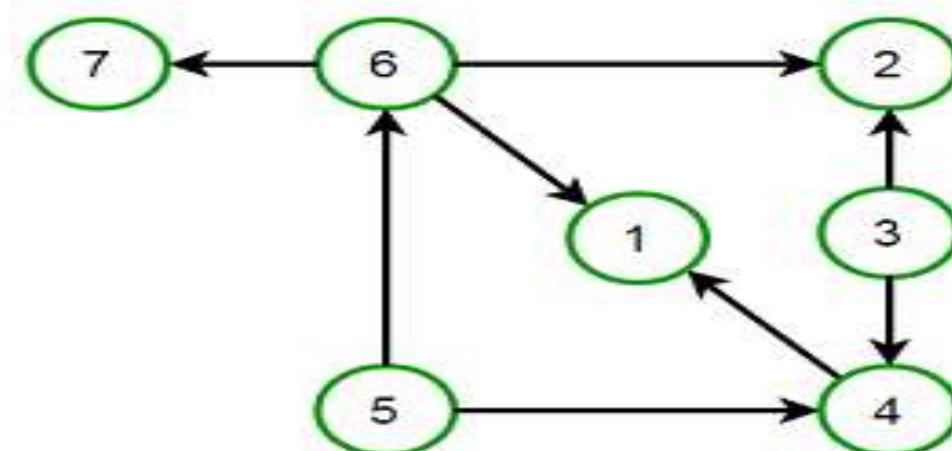


Graph Terminology

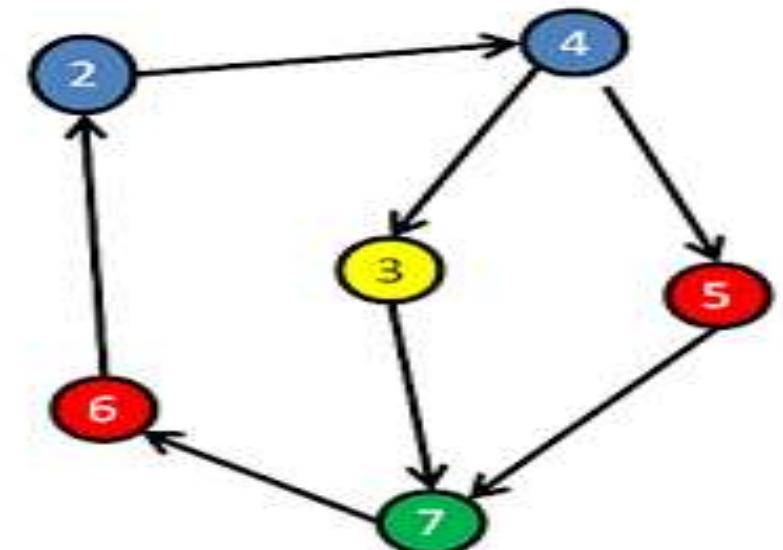
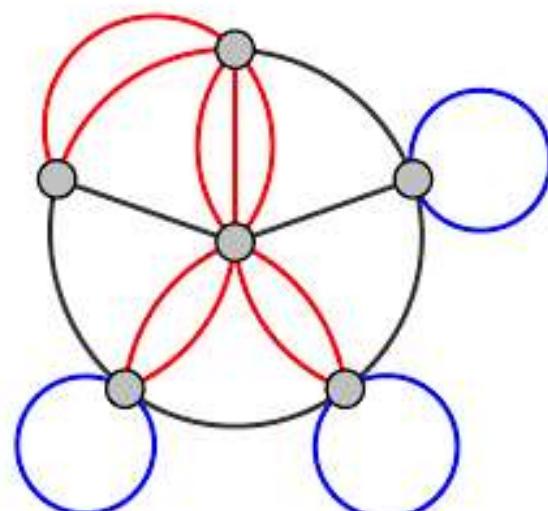
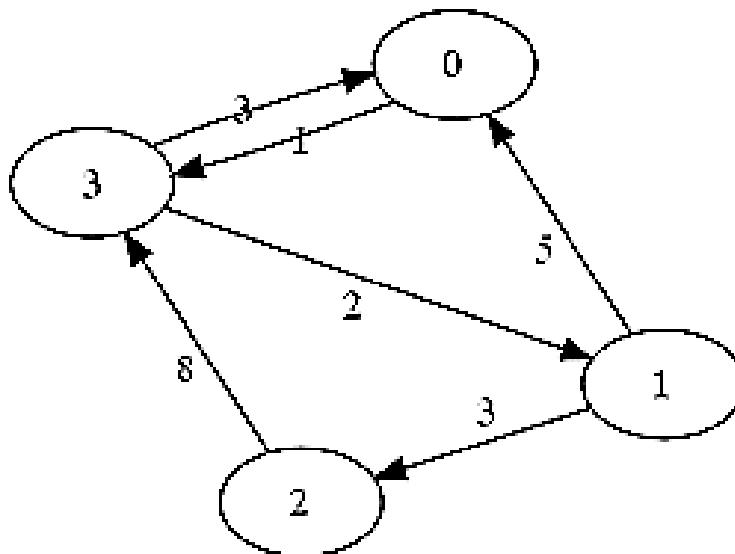
- **Undirected Graph:-** A graph with only undirected edges is said to be undirected graph.
- **Directed Graph:-** A graph with only directed edges is said to be directed graph.
- **Mixed Graph:-** A graph with undirected and directed edges is said to be mixed graph.
- **End vertices or Endpoints:-** The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.



- **Origin:-** If an edge is directed, its first endpoint is said to be origin of it.
- **Destination:-** If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.
- **Adjacent:-** If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.
- **Incident:-** An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.



- **Outgoing Edge**:-A directed edge is said to be outgoing edge on its origin vertex.
- **Incoming Edge**:-A directed edge is said to be incoming edge on its destination vertex.
- **Degree**:-Total number of edges connected to a vertex is said to be degree of that vertex.
- **In-degree**:-Total number of incoming edges connected to a vertex is said to be in-degree of that vertex.
- **Out degree**:-Total number of outgoing edges connected to a vertex is said to be out degree of that vertex.
- **Parallel edges or Multiple edges**:-If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.



➤ **Self-loop:-** An edge (undirected or directed) is a self-loop if its two endpoints coincide.

➤ **Simple Graph:-** A graph is said to be simple if there are no parallel and self-loop edges.

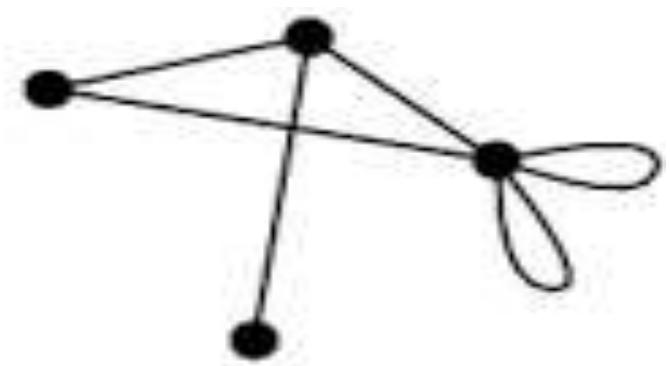
➤ **Path:-** A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.



simple graph



*nonsimple graph
with multiple edges*



*nonsimple graph
with loops*

Graph Representations

Graph data structure is represented using following representations...

➤ **Adjacency Matrix**

➤ **Incidence Matrix**

➤ **Adjacency List**

a. **Adjacency Matrix:-**

✓ In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices.

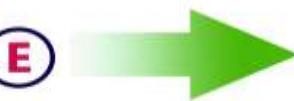
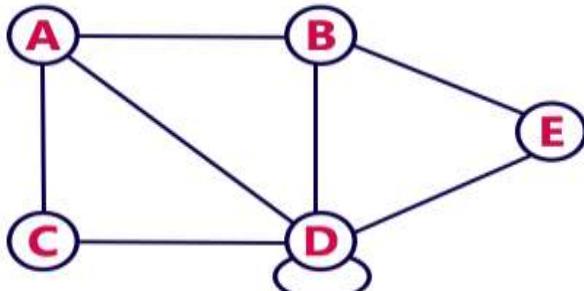
That means if a graph with 4 vertices can be represented using a matrix of 4X4 class.

✓ In this matrix, rows and columns both represents vertices.

✓ This matrix is filled with either 1 or 0.

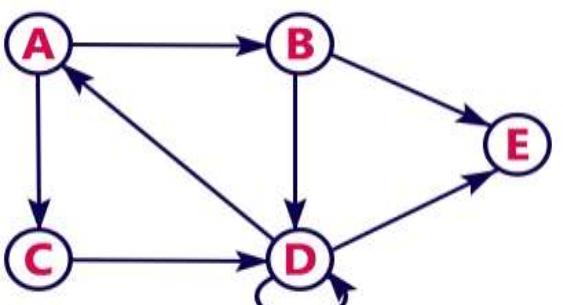
✓ Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

- For example, consider the following undirected graph representation...



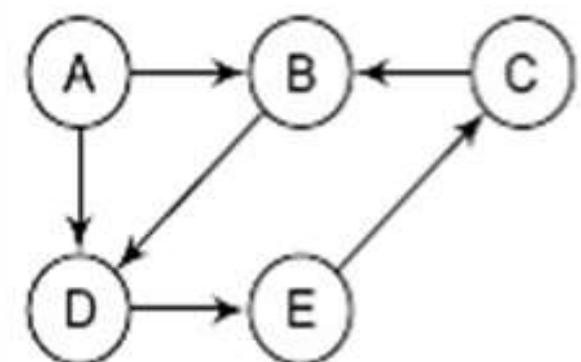
	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Directed graph representation...



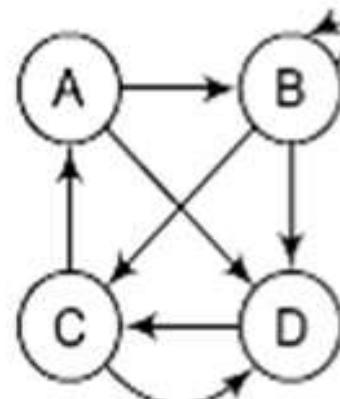
	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

Adjacency Matrix Representation



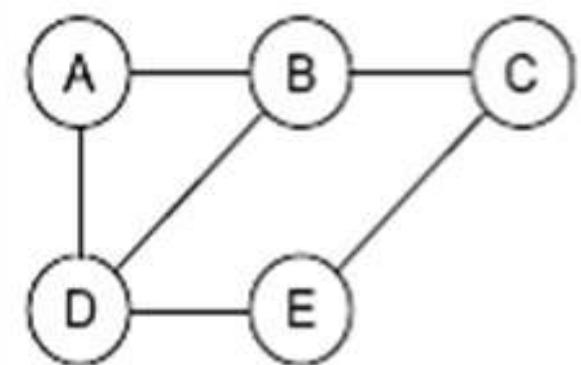
	A	B	C	D	E
A	0	1	0	1	0
B	0	0	0	1	0
C	0	1	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

(a) Directed graph



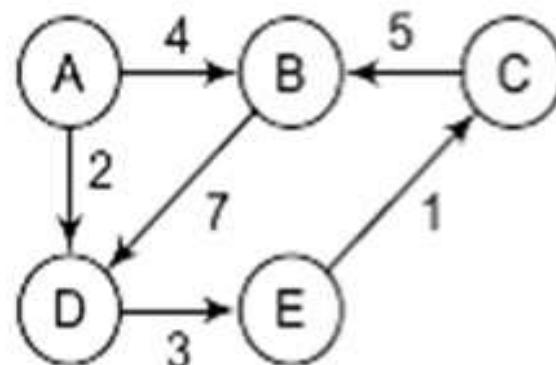
	A	B	C	D
A	0	1	0	1
B	0	1	1	1
C	1	0	0	1
D	0	0	1	0

(b) Directed graph with loop



	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

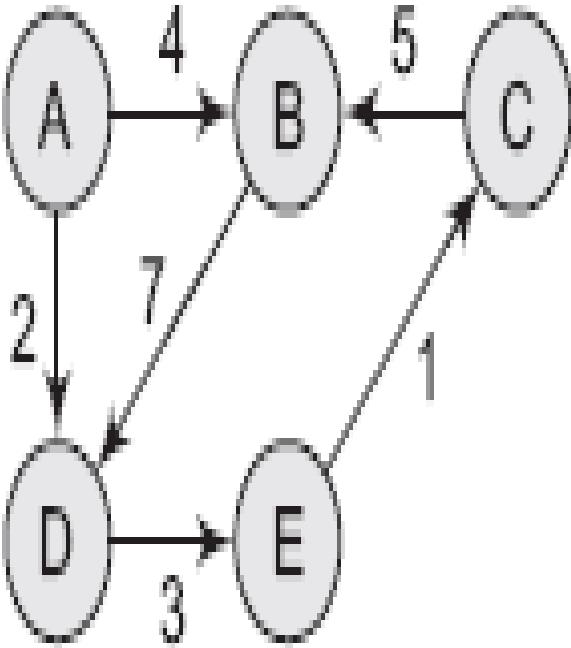
(c) Undirected graph



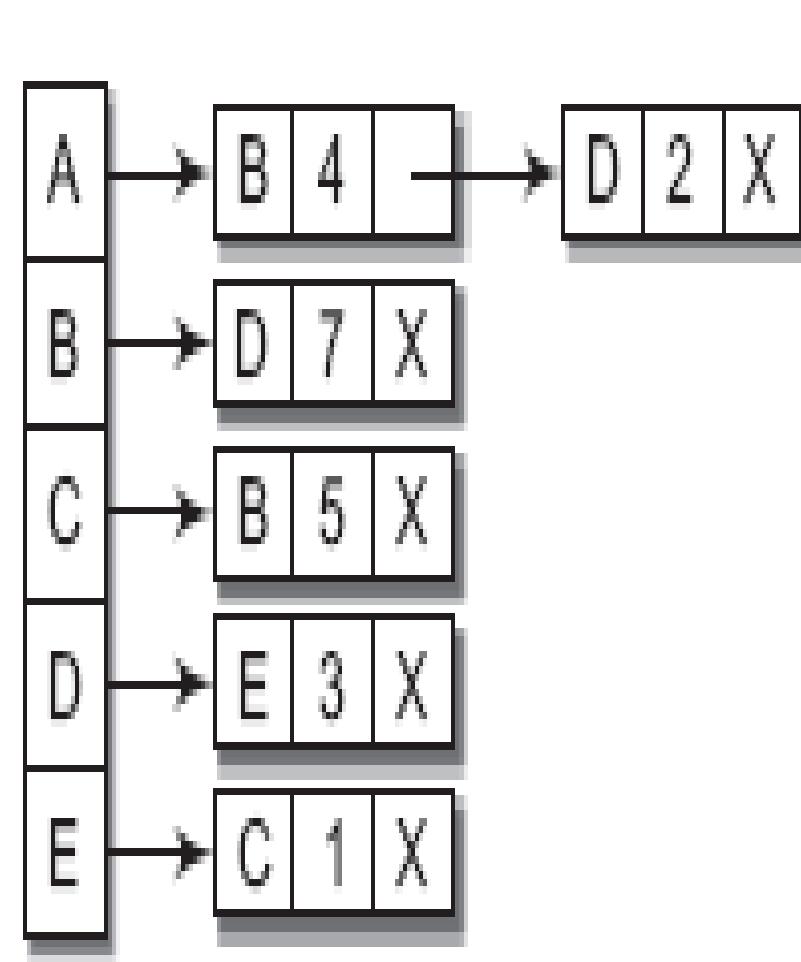
	A	B	C	D	E
A	0	4	0	2	0
B	0	0	0	7	0
C	0	5	0	0	0
D	0	0	0	0	3
E	0	0	1	0	0

(d) Weighted graph

Adjacency Matrix Representation



(Weighted graph)

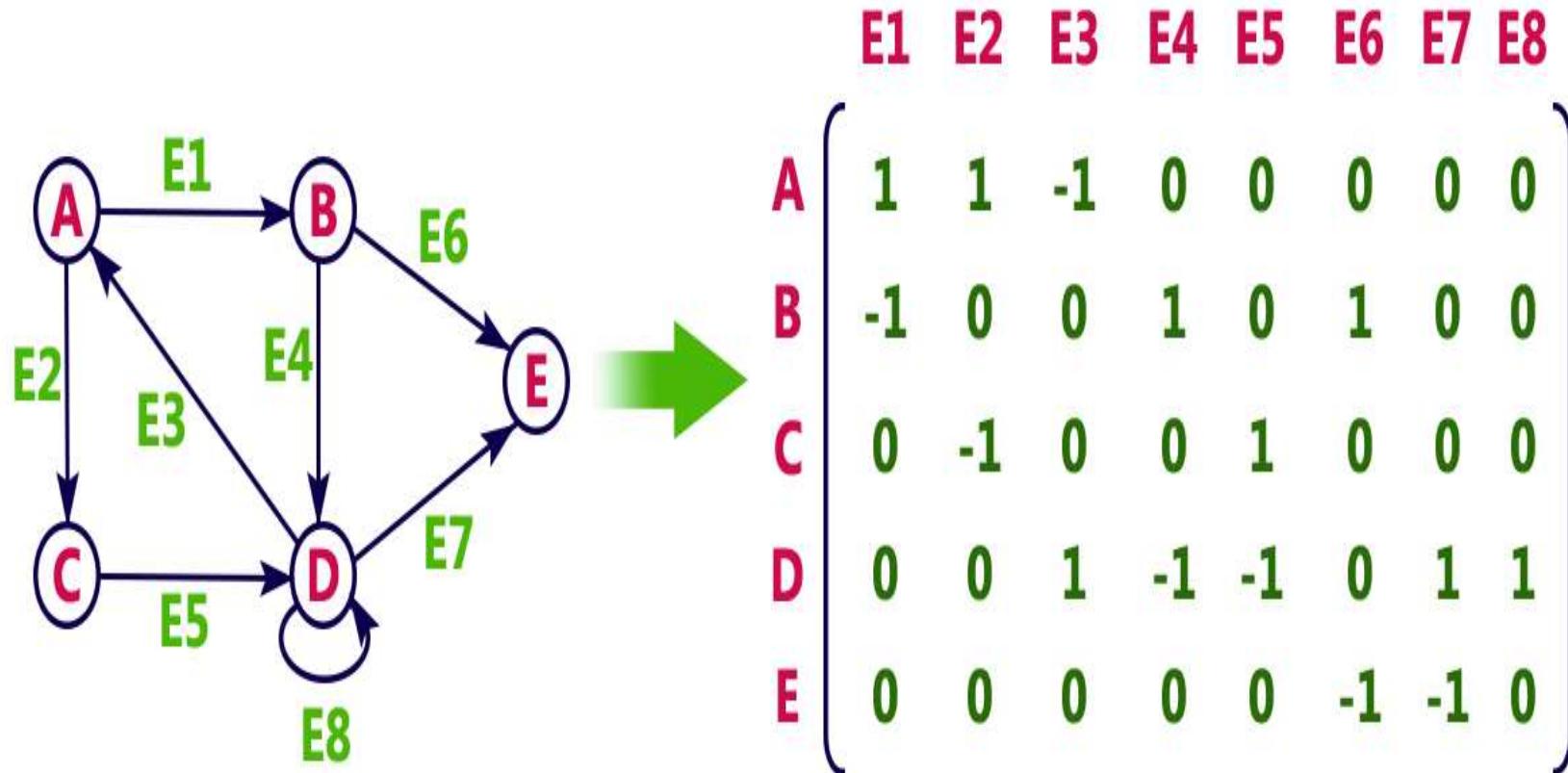


Graph Representations

Incidence Matrix

- In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges.
- That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represents vertices and columns represents edges.
- This matrix is filled with either 0 or 1 or -1.
- Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

- For example, consider the following directed graph representation...

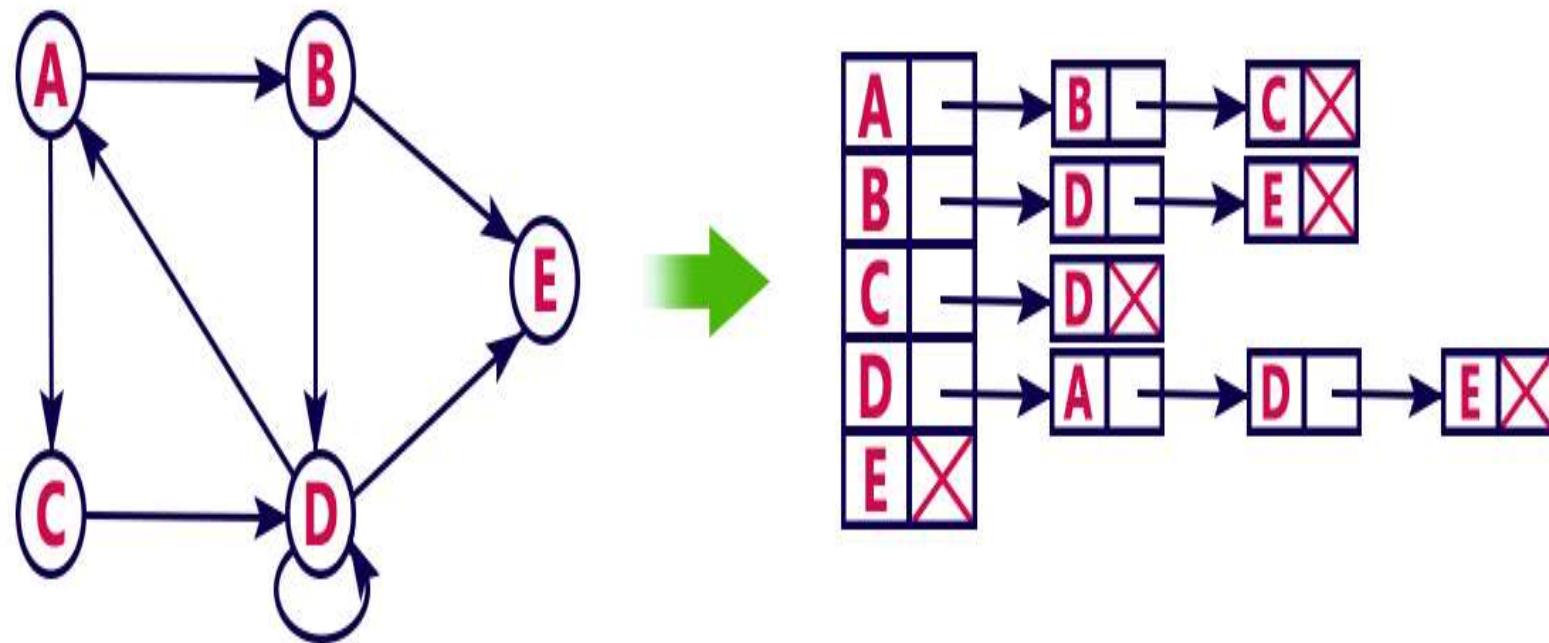


Graph Representations

Adjacency List

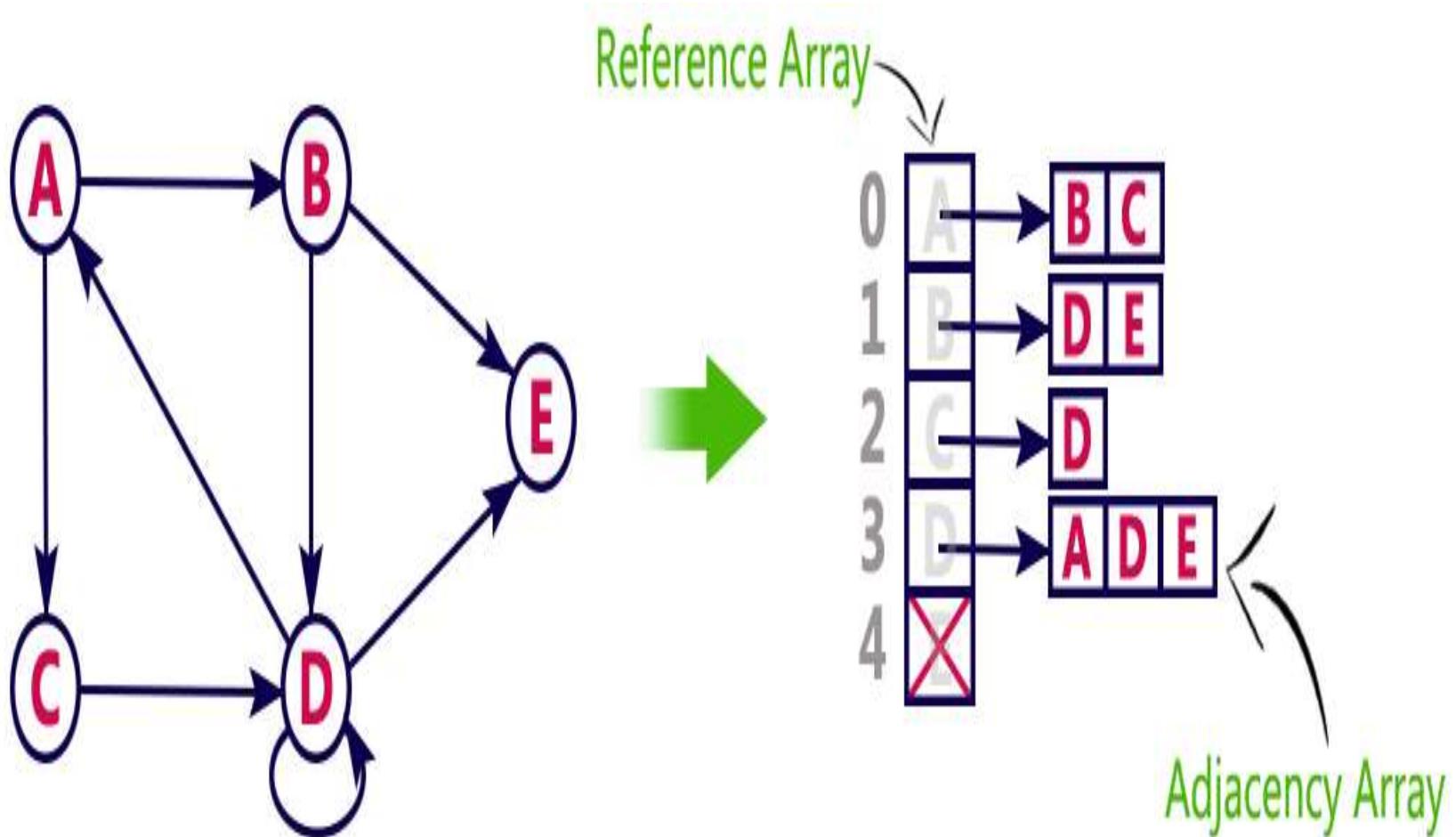
- In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



Graph Array Representations

- This representation can also be implemented using array as follows..



Operations of Graphs

- INSERTING
- DELETING
- MERGING
- TRAVERSING

- **Graph traversal is technique used for searching a vertex in a graph.** The graph traversal is also used to decide the order of vertices to be visit in the search process.
- **A graph traversal finds the edges to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.**

There are two graph traversal techniques and they are as follows...

➤ **DFS (Depth First Search)**

➤ **BFS (Breadth First Search)**

Depth First Search

DFS (Depth First Search)

- DFS traversal of a graph, produces a **spanning tree** as final result.
- **Spanning Tree** is a graph without any loops.
- We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

DFS (Depth First Search)

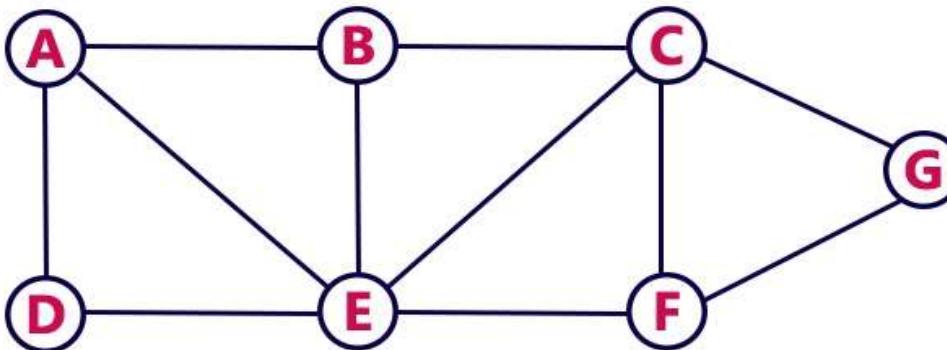
Algorithm

- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

EXAMPLE

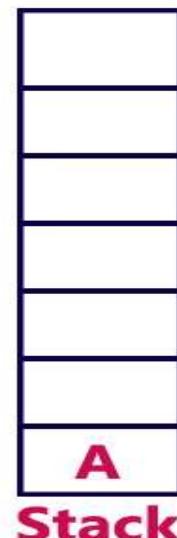
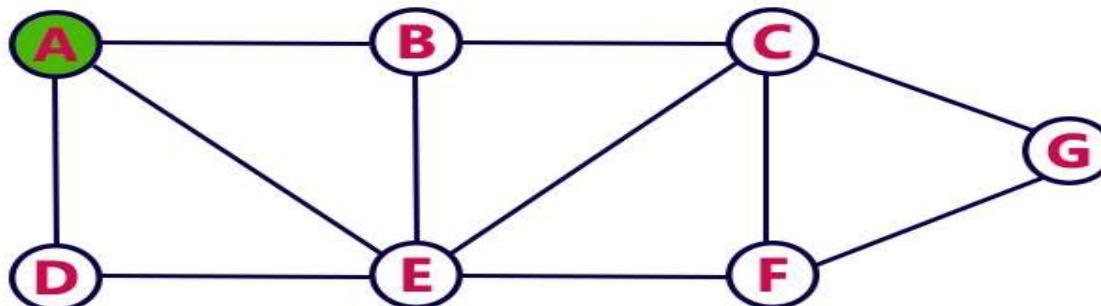
- Back tracking is coming back to the vertex from which we came to current vertex.

Consider the following example graph to perform DFS traversal



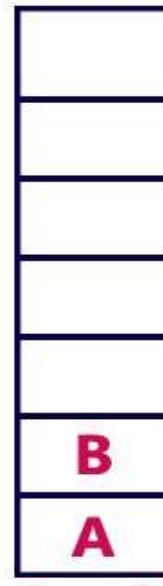
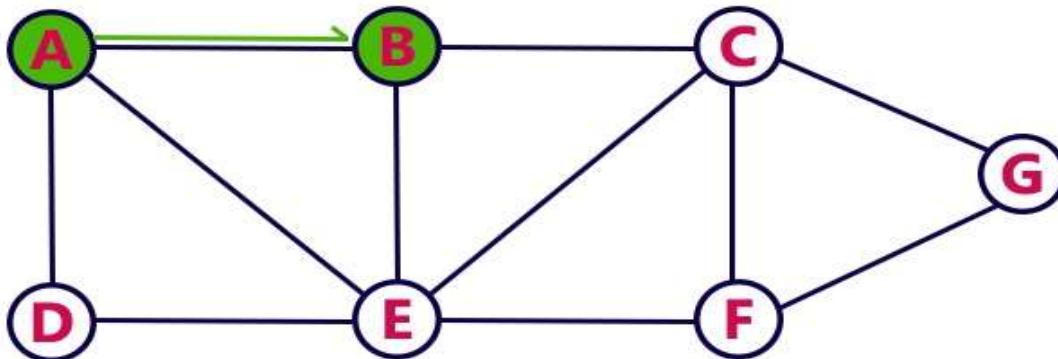
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



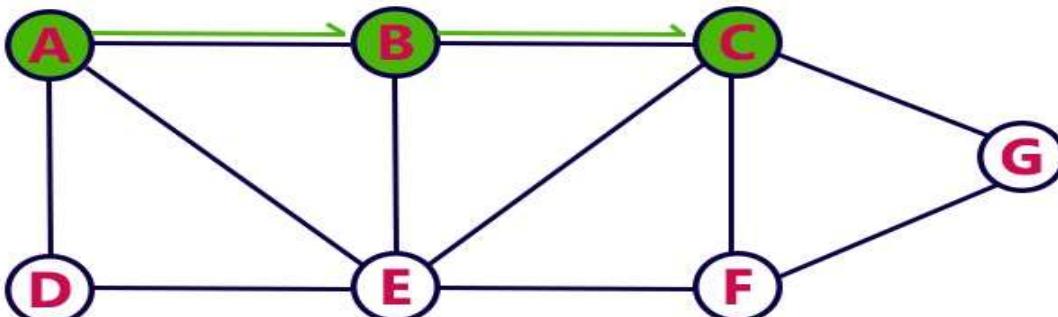
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



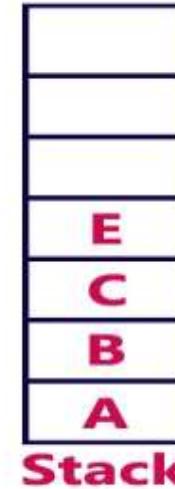
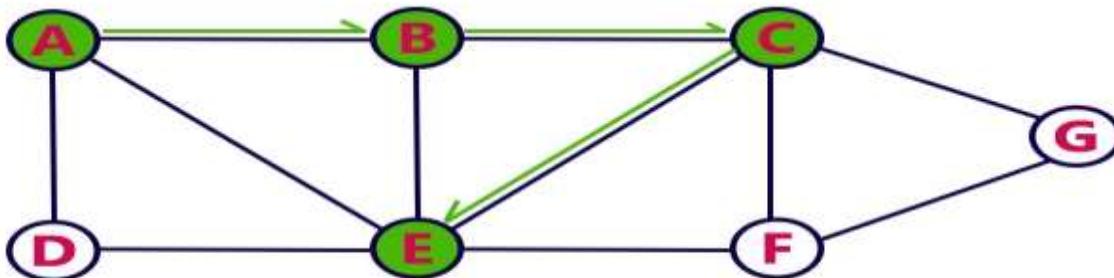
Step 3:

- Visit any adjacent vertex of **B** which is not visited (**C**).
- Push C on to the Stack.



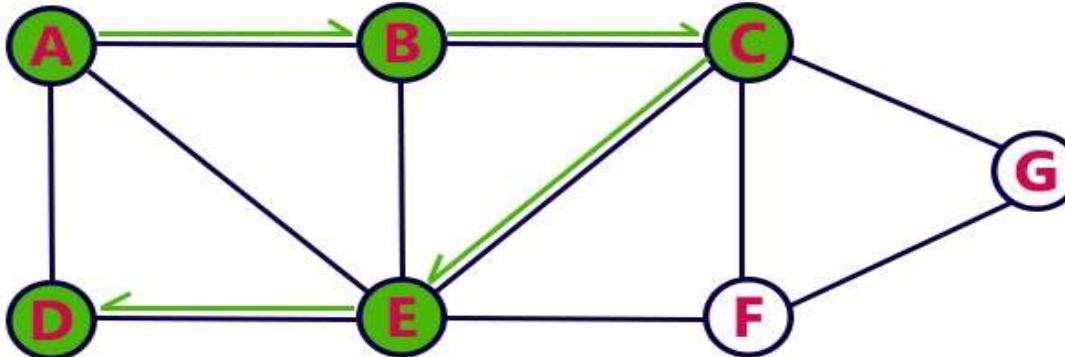
Step 4:

- Visit any adjacent vertex of **C** which is not visited (**E**).
- Push E on to the Stack



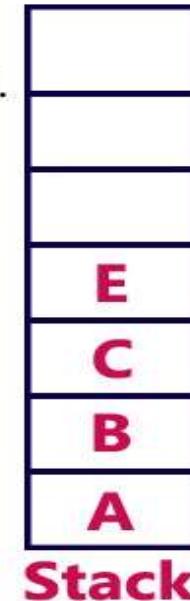
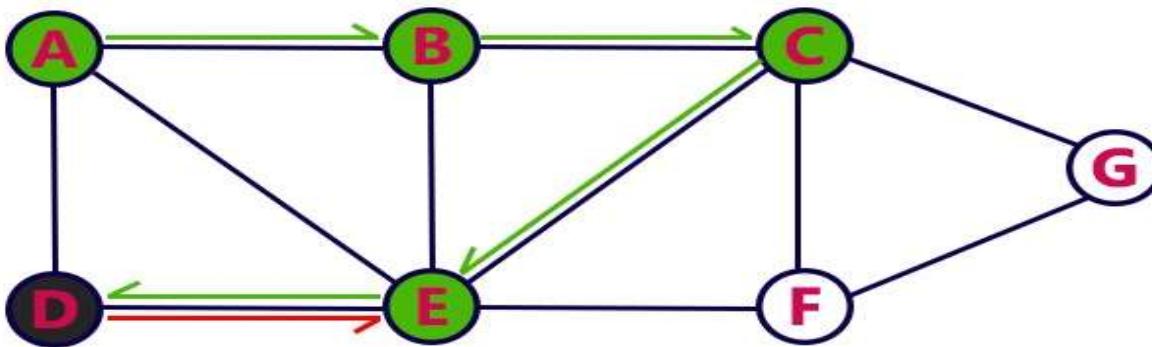
Step 5:

- Visit any adjacent vertex of **E** which is not visited (**D**).
- Push D on to the Stack



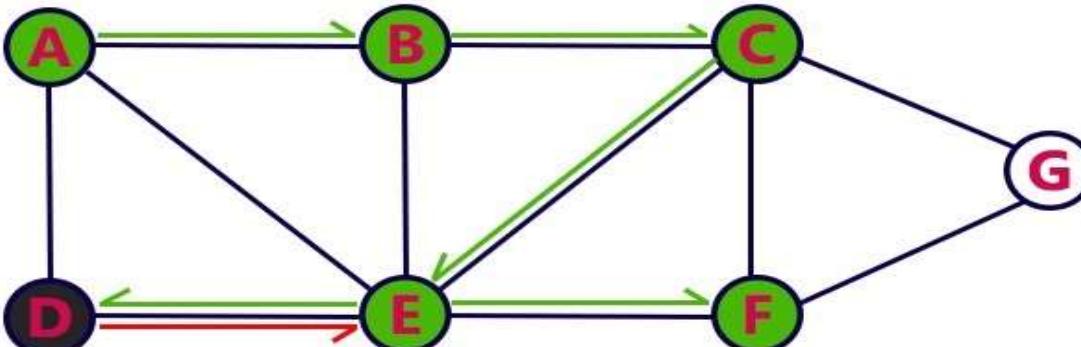
Step 6:

- There is no new vertex to be visited from D. So use back track.
- Pop D from the Stack.



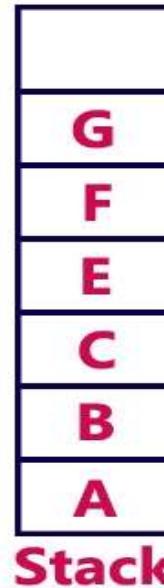
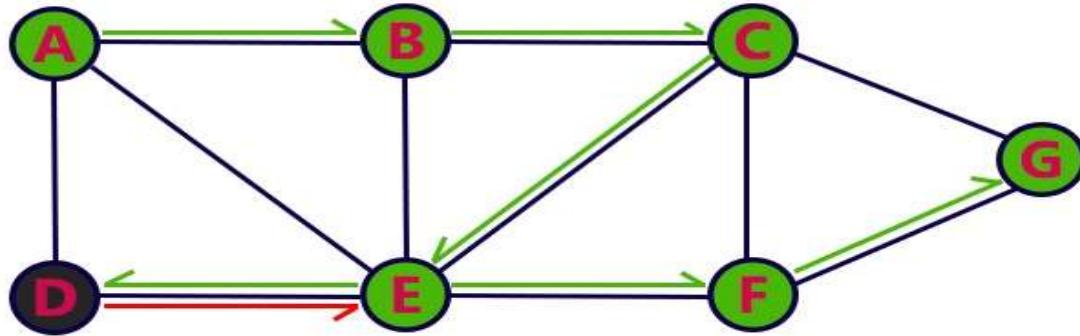
Step 7:

- Visit any adjacent vertex of E which is not visited (F).
- Push F on to the Stack.



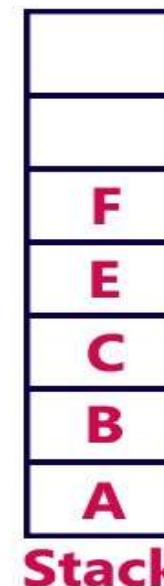
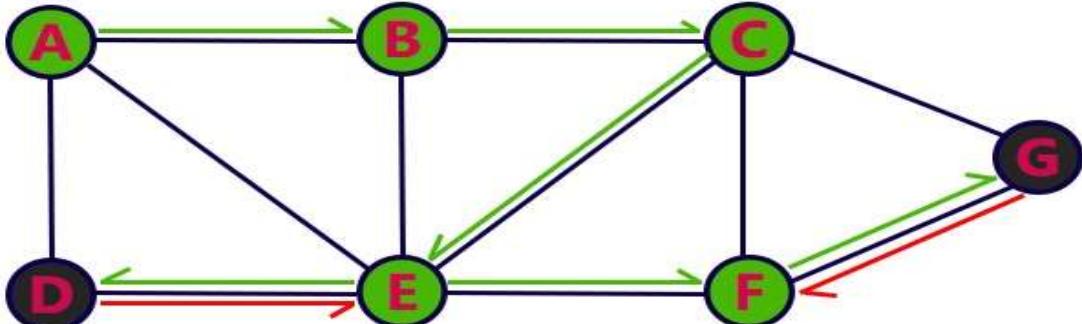
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



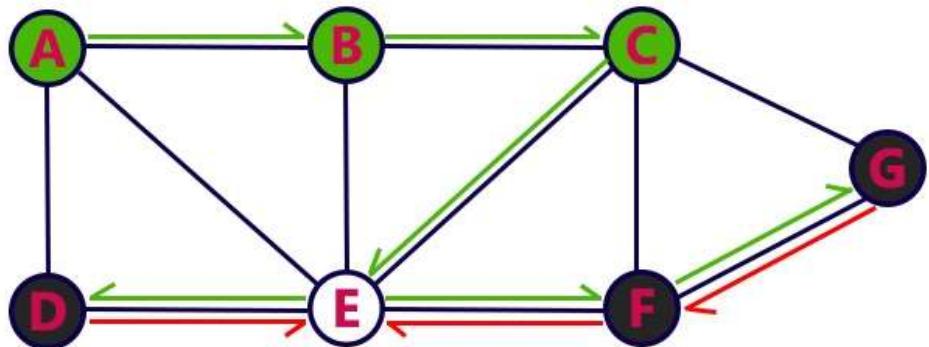
Step 9:

- There is no new vertex to be visited from G. So use back track.
- Pop G from the Stack.



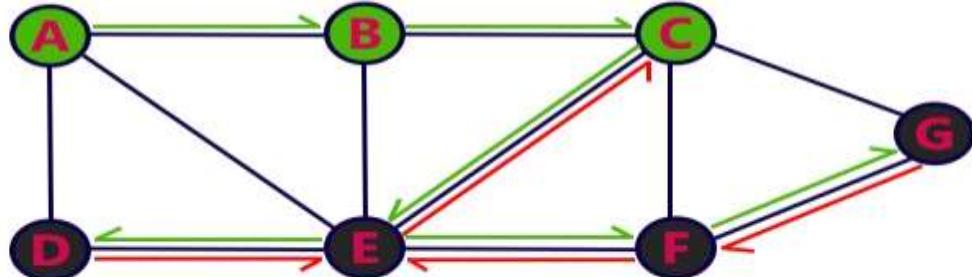
Step 10:

- There is no new vertex to be visited from F. So use back track.
- Pop F from the Stack.



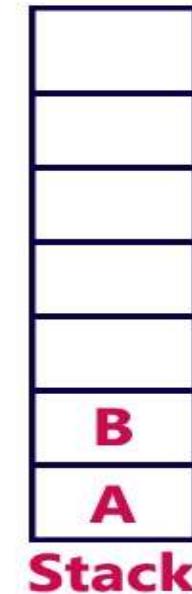
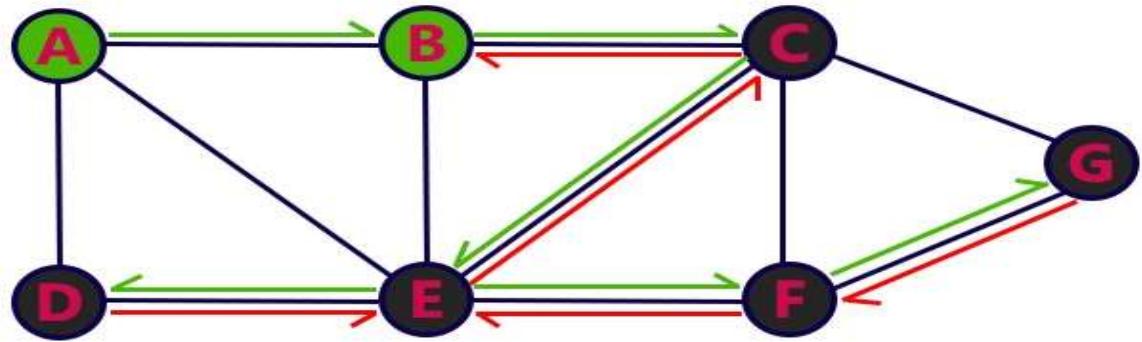
Step 11:

- There is no new vertex to be visited from E. So use back track.
- Pop E from the Stack.



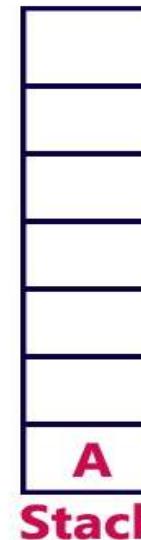
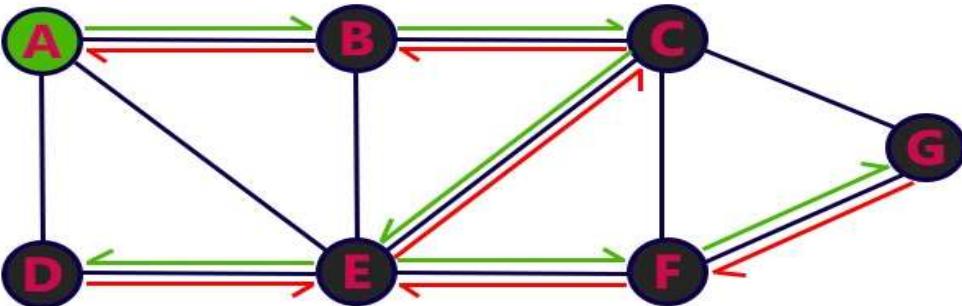
Step 12:

- There is no new vertex to be visited from C. So use back track.
- Pop C from the Stack.



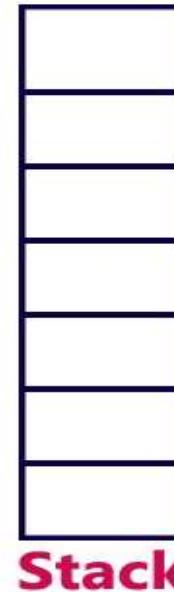
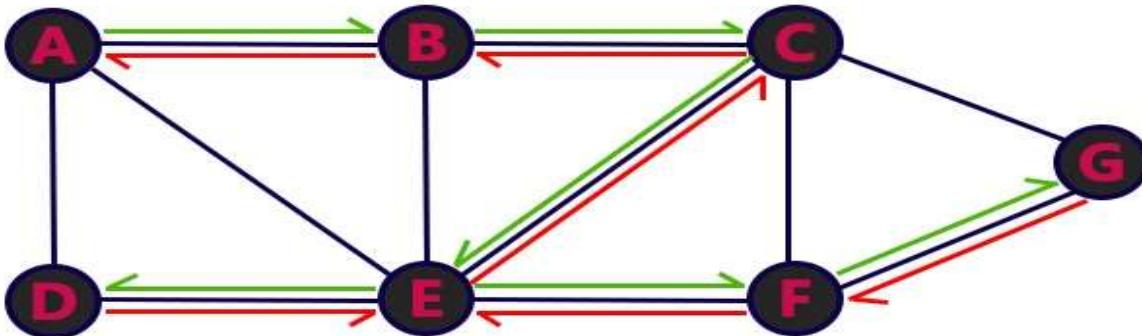
Step 13:

- There is no new vertex to be visited from B. So use back track.
- Pop B from the Stack.

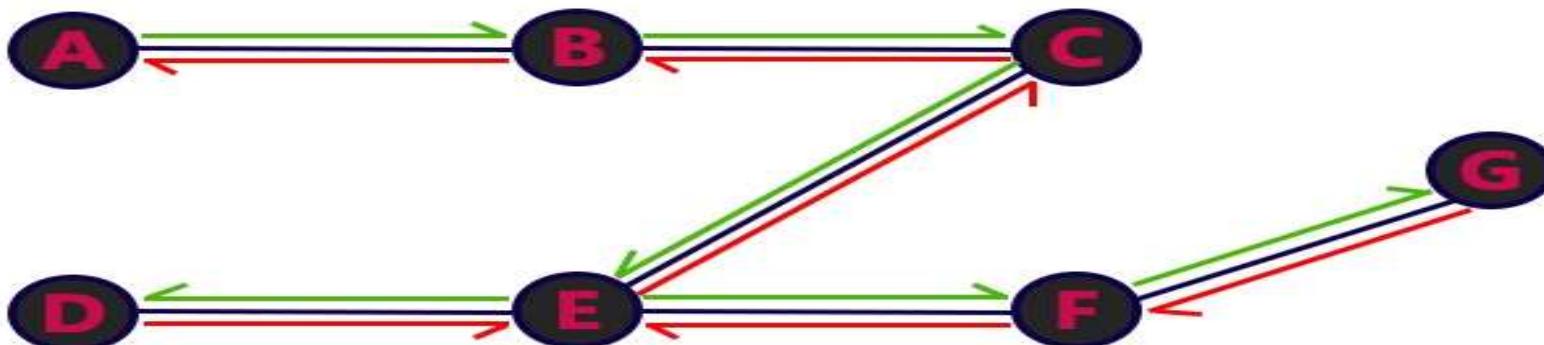


Step 14:

- There is no new vertex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Traversal.
- Final result of DFS traversal is following spanning tree.



Breadth First Search

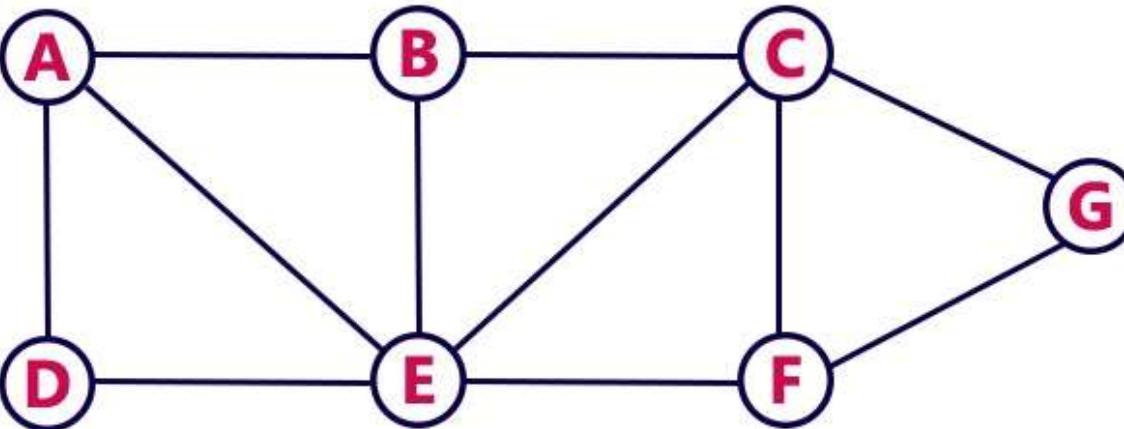
- ✓ BFS traversal of a graph, produces a **spanning tree** as final result.
- ✓ **Spanning Tree** is a graph without any loops.
- ✓ We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

BFS (Breadth First Search)

➤ ALGORITHM

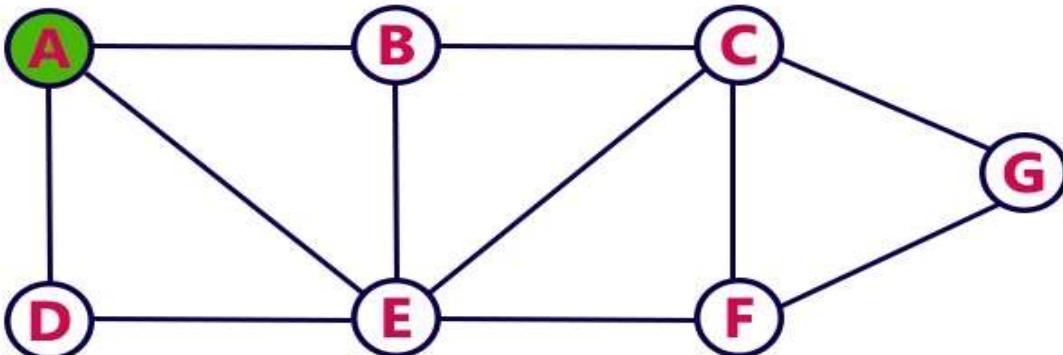
- ✓ **Step 1:** Define a Queue of size total number of vertices in the graph.
- ✓ **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- ✓ **Step 3:** Visit all the **adjacent** vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.
- ✓ **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- ✓ **Step 5:** Repeat step 3 and 4 until queue becomes empty.
- ✓ **Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph.

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

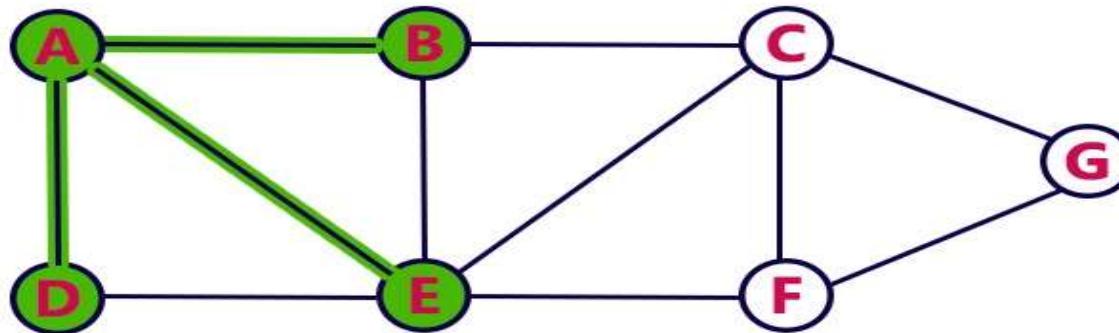


Queue



Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D, E, B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

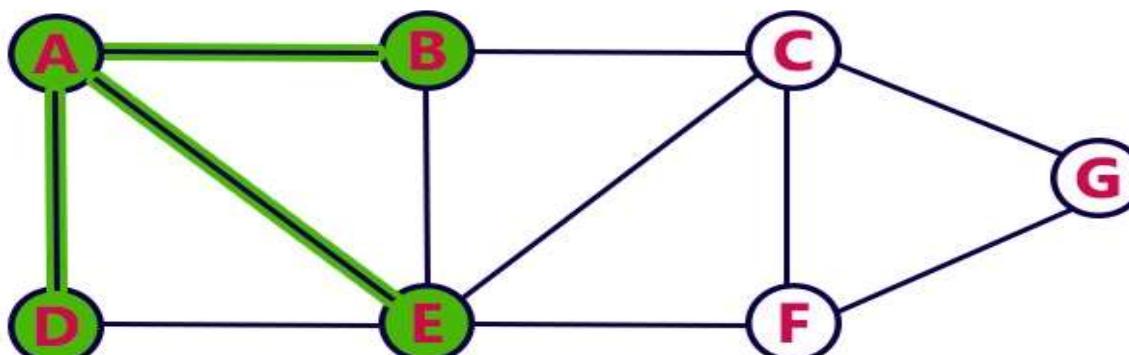


Queue



Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

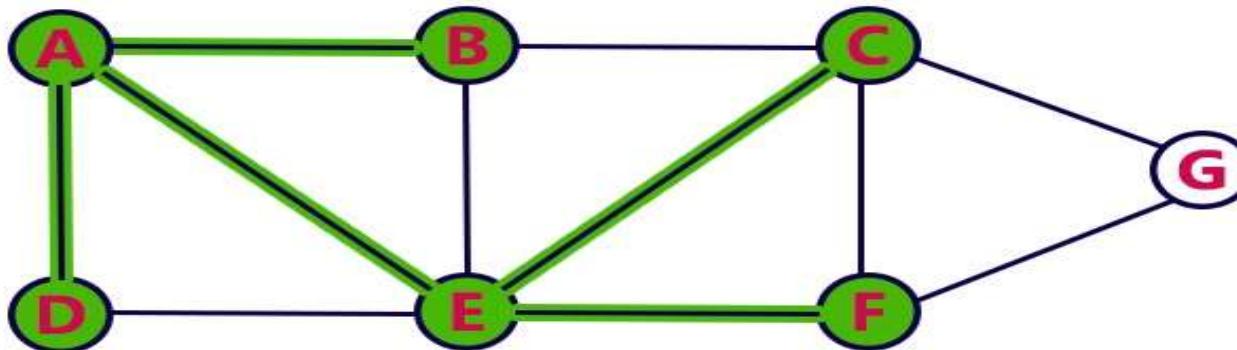


Queue



Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

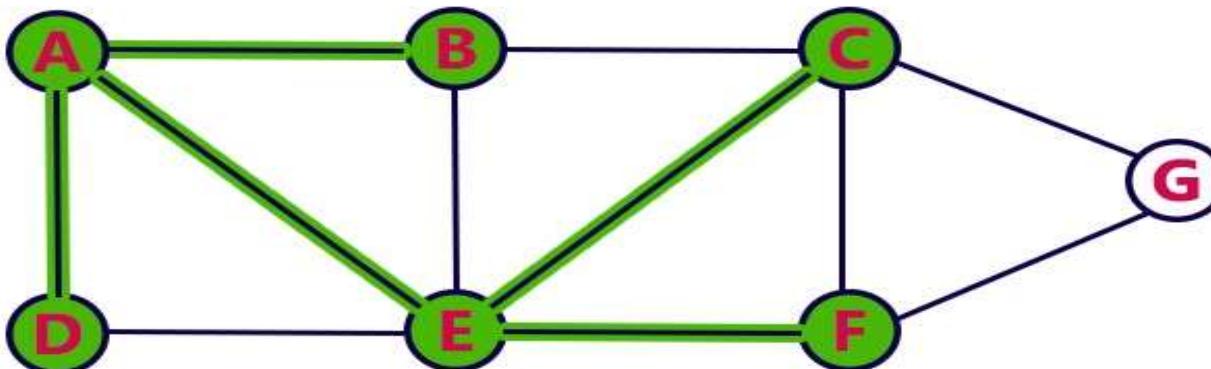


Queue



Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

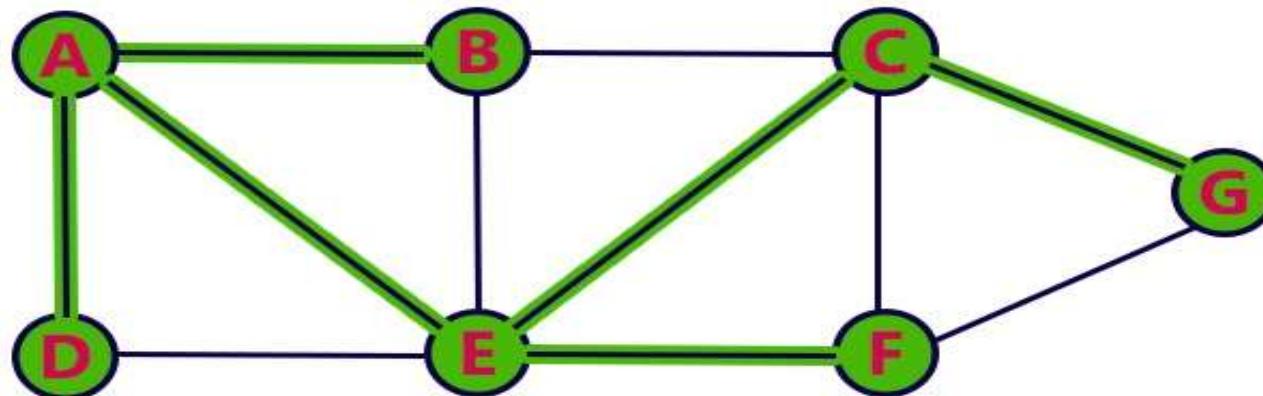


Queue

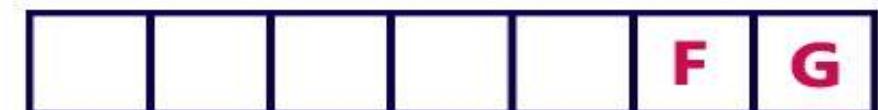


Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

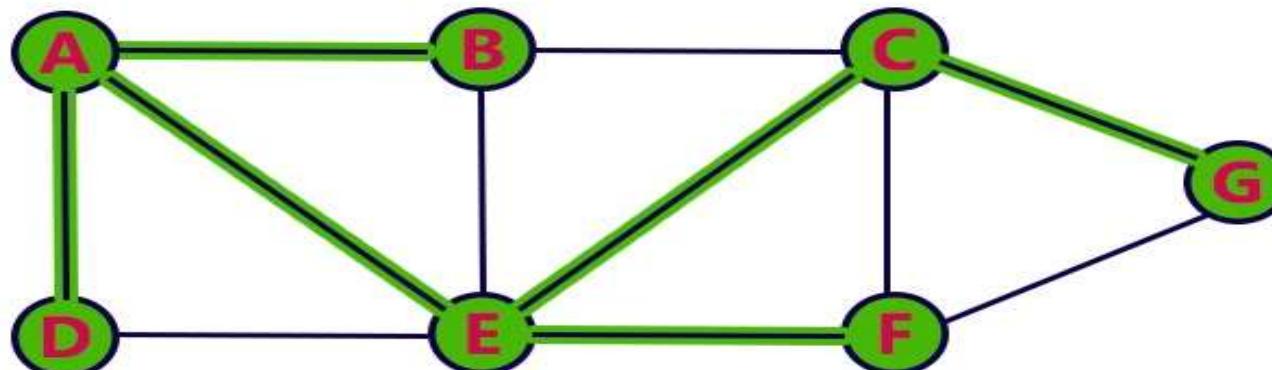


Queue



Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

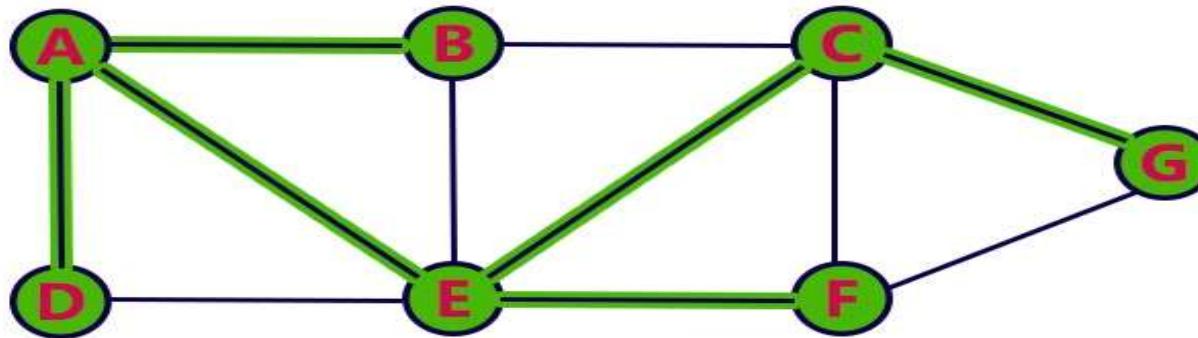


Queue



Step 8:

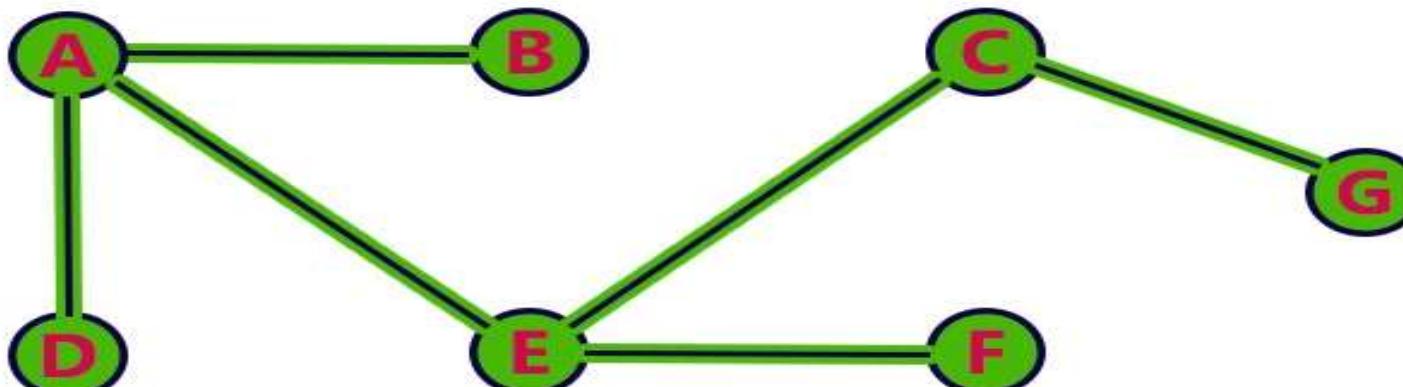
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Shortest Path Algorithm

Spanning tree

- A spanning tree is a sub-graph of an undirected and a connected graph, which includes all the vertices of the graph having a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.
- The edges may or may not have weights assigned to them.
- The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n(n-2)$.

Shortest Path Algorithm

In this section we will discuss the three different algorithms to calculate the shortest path in between the vertices of the graph

➤ Minimum Spanning Trees

Prim's Algorithm

Kruskal's Algorithm

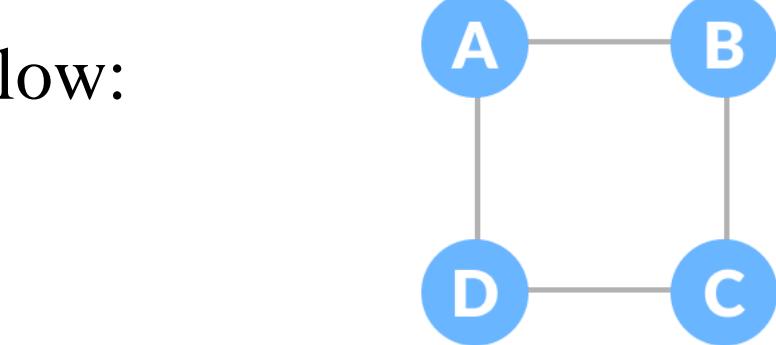
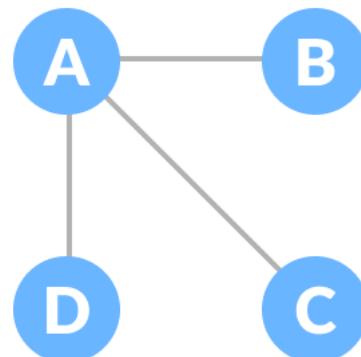
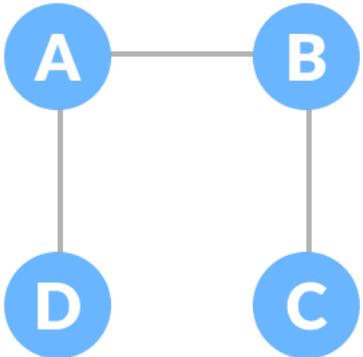
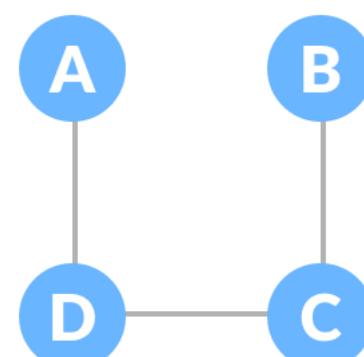
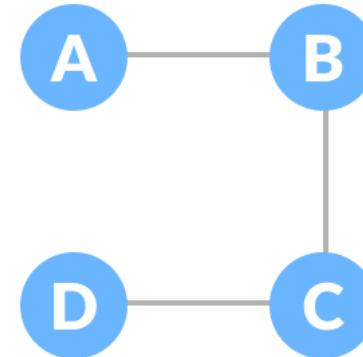
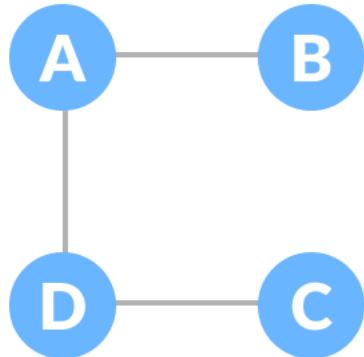
Minimum Spanning Trees

- ✓ A spanning tree of a connected, undirected graph G , is a sub-graph of G which is a tree that connects all the vertices together. A graph G can have many different spanning trees.
- ✓ When we assign weights to each edge (which is a number that represents how unfavorable the edge is), and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning tree.
- ✓ A minimum spanning tree (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. In other words, a **minimum spanning tree** is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

Example of a Spanning Tree

Let's understand the spanning tree with examples below:

Let the original graph be:

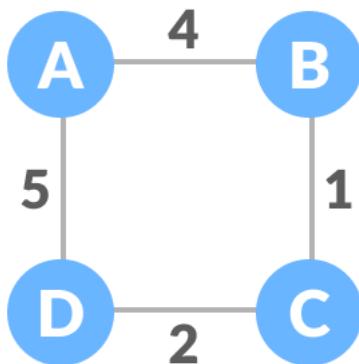


Minimum Spanning Tree

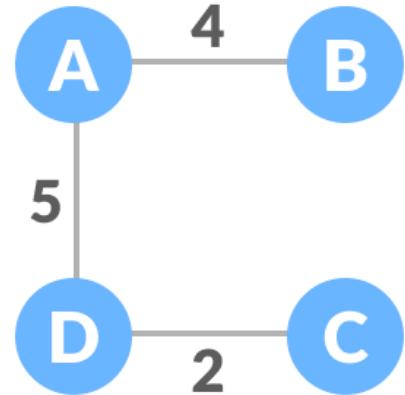
A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

Example of a Spanning Tree

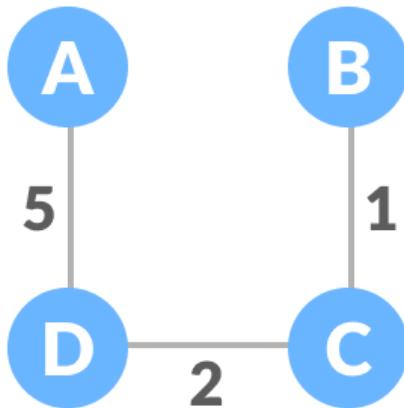
Let's understand the above definition with the help of the example below.
The initial graph is:



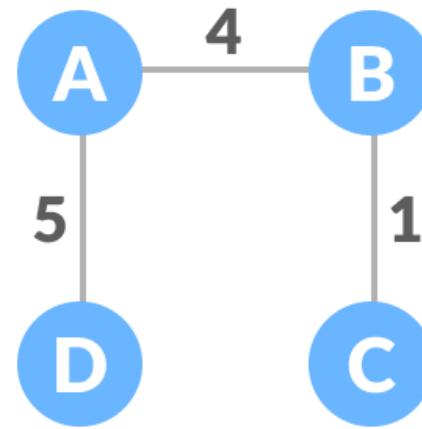
The possible spanning trees are:



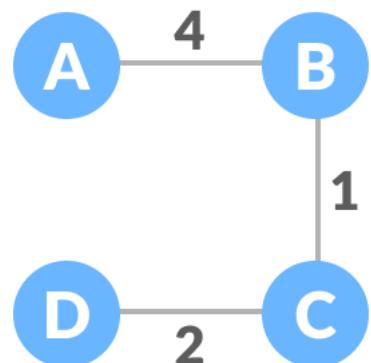
$$\text{sum} = 11$$



$$\text{sum} = 8$$



$$\text{sum} = 10$$



$$\text{sum} = 7$$

Applications of MST

- ✓ MST widely used for designing networks.
- ✓ MST is used to determine the least costly path with no cycles in this network.
- ✓ MST are used to find airline routers.
- ✓ MST are also used to find the cheapest way to connect terminals.
- ✓ MST are applied in routing algorithms for finding the most efficient path.

PRIM'S ALGORITHM

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which form a tree that includes every vertex has the minimum sum of weights among all the trees that can be formed from the graph

How Prim's algorithm works

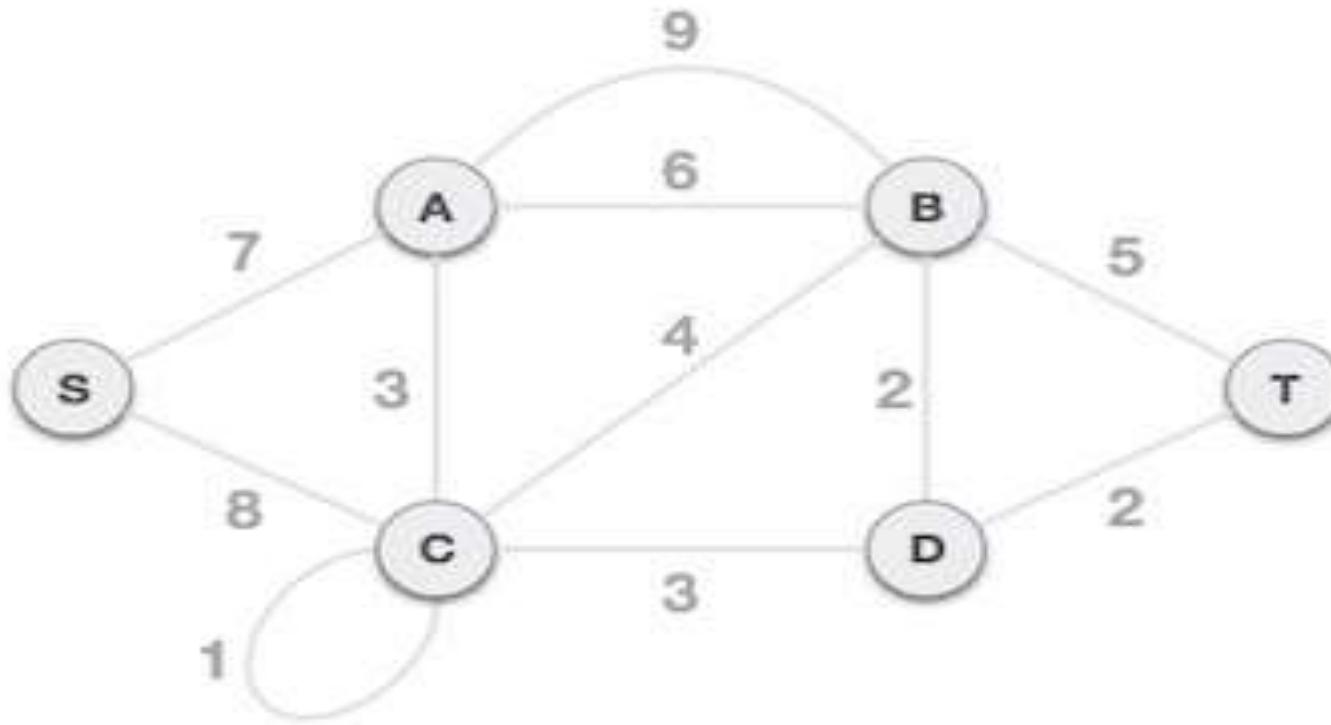
It falls under a class of algorithms called greedy algorithms which find the local optimum in the hopes of finding a global optimum.

- We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

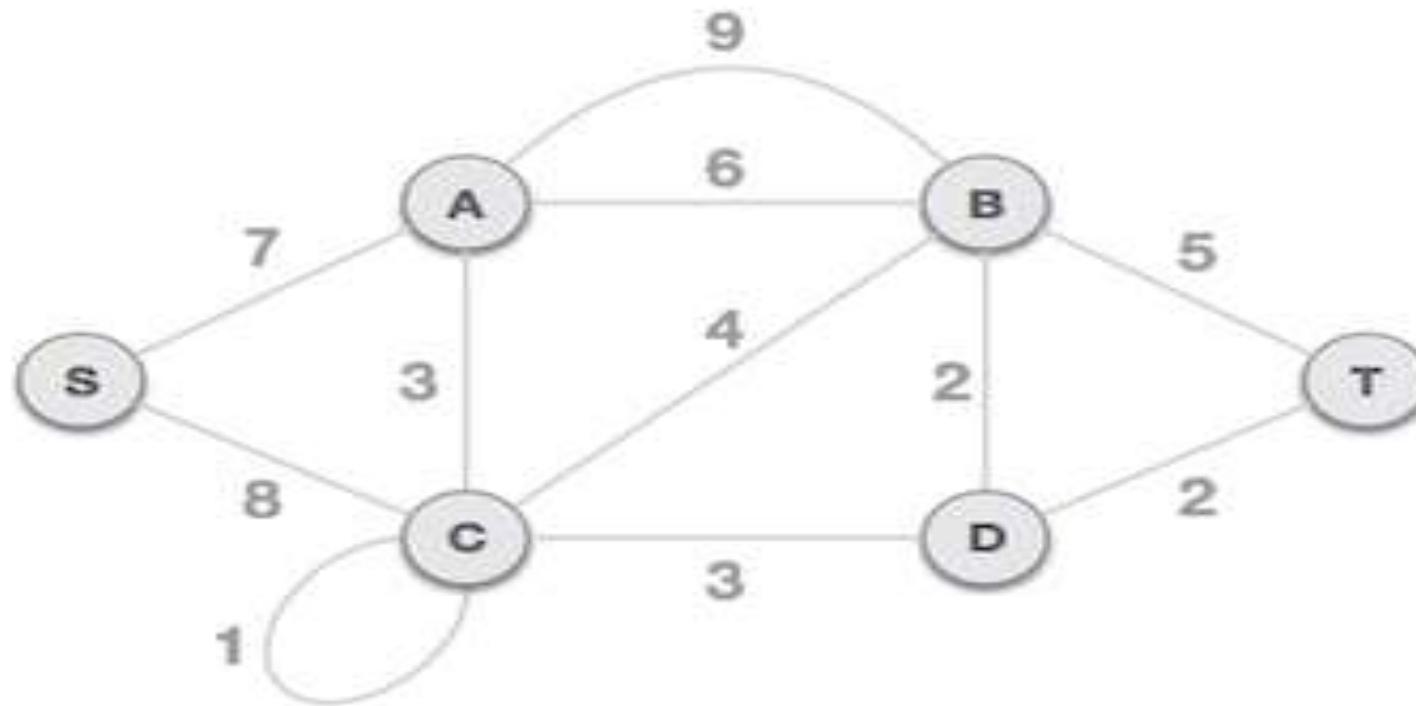
- Initialize the minimum spanning tree with a vertex chosen at random.
- Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
- Keep repeating step 2 until we get a minimum spanning tree

PRIM'S ALGORITHM



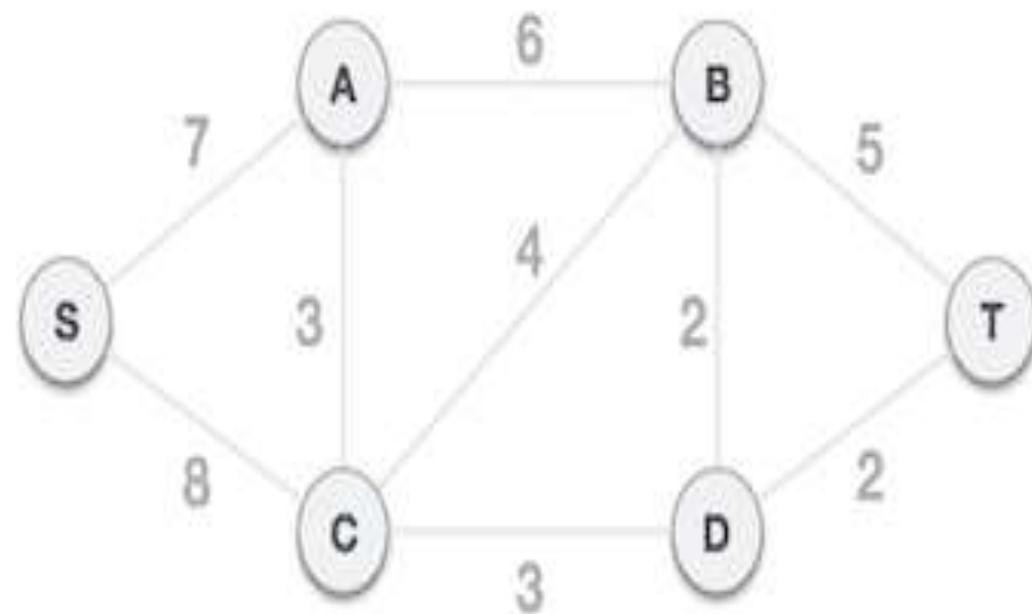
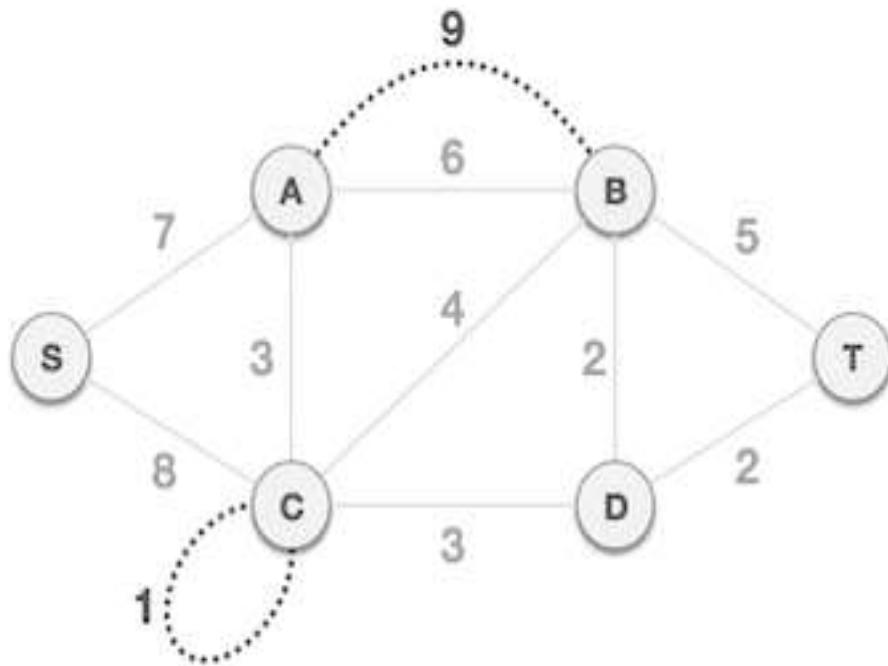
PRIM'S ALGORITHM

- Step 1: Select a starting vertex
- Step 2: Repeat Steps 3 and 4 until there are fringe vertices
- Step 3: Select an edge e connecting the tree vertex and fringe vertex that has minimum weight
- Step 4: Add the selected edge and the vertex to the minimum spanning tree T

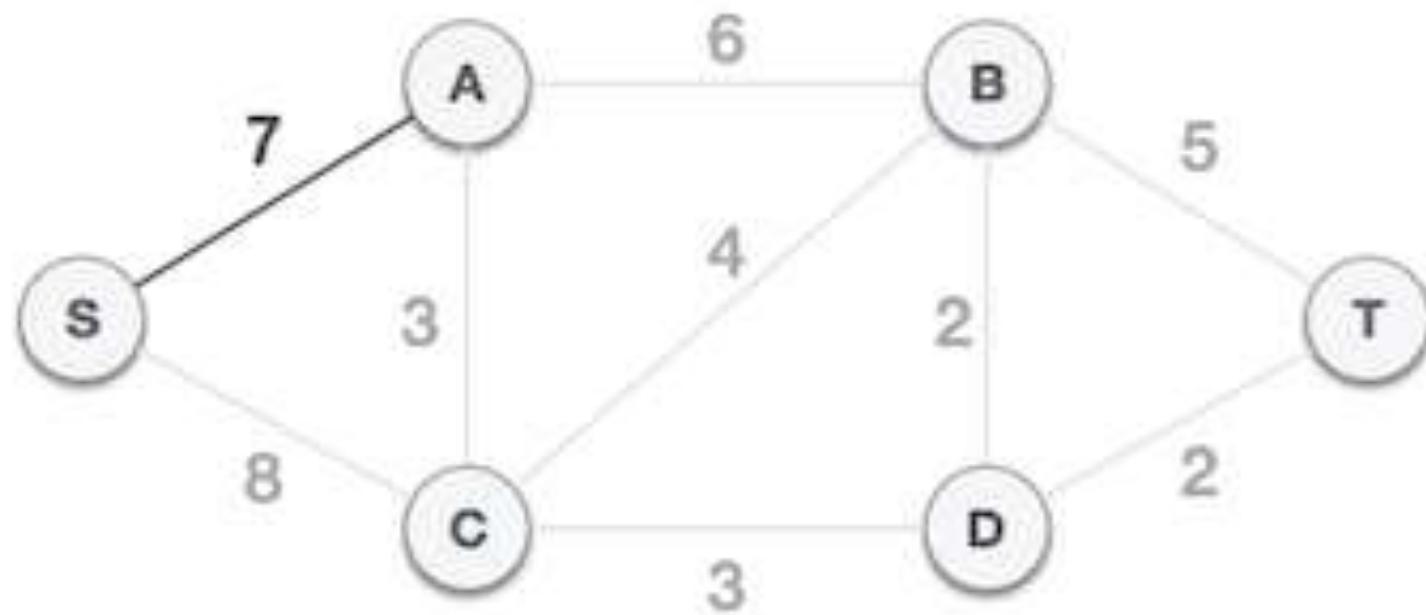


- Step 1 Remove all loops and parallel edges

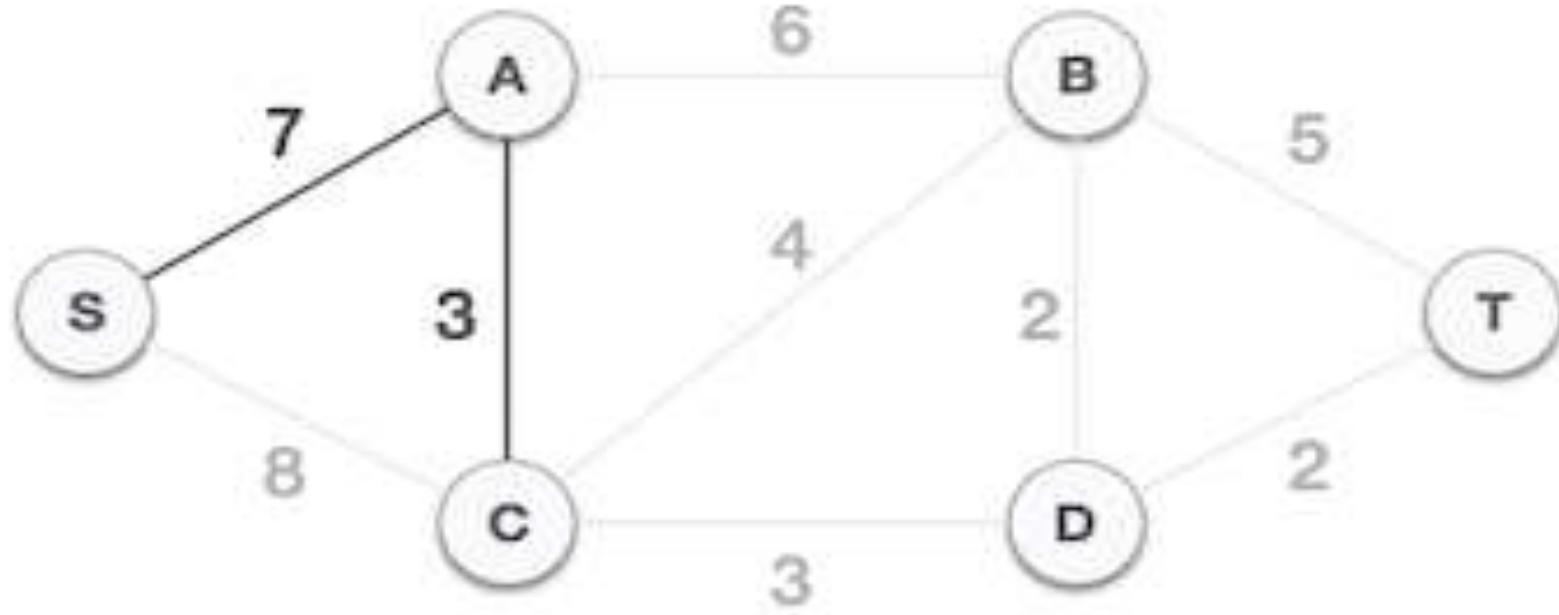
Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.



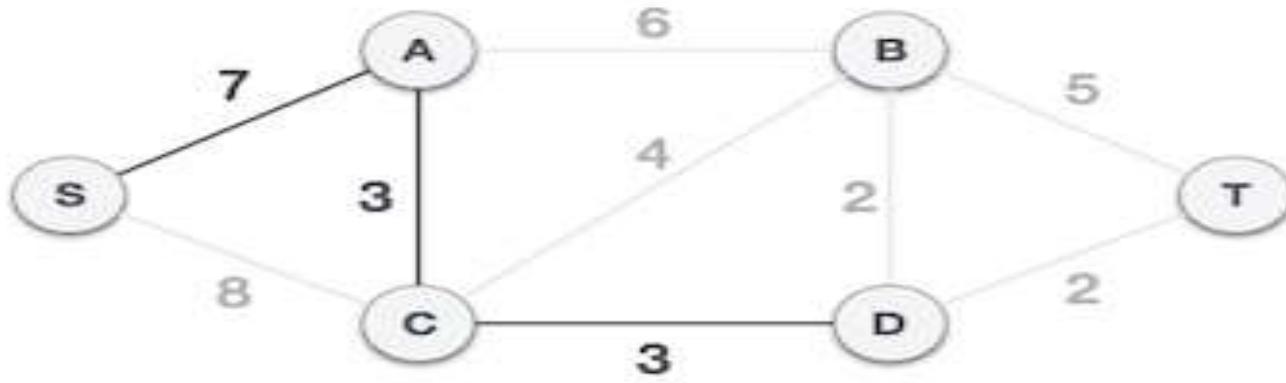
- Step 2 Choose any arbitrary node as root node
- In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.
- Step 3 Check outgoing edges and select the one with less cost
- After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



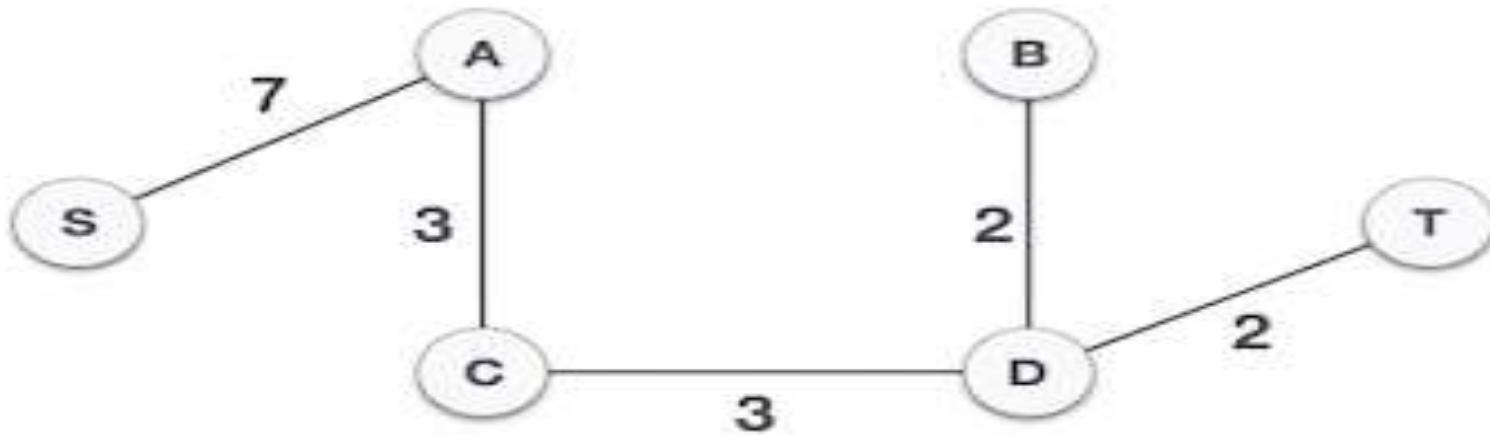
- Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



- After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



Kruskal's ALGORITHM

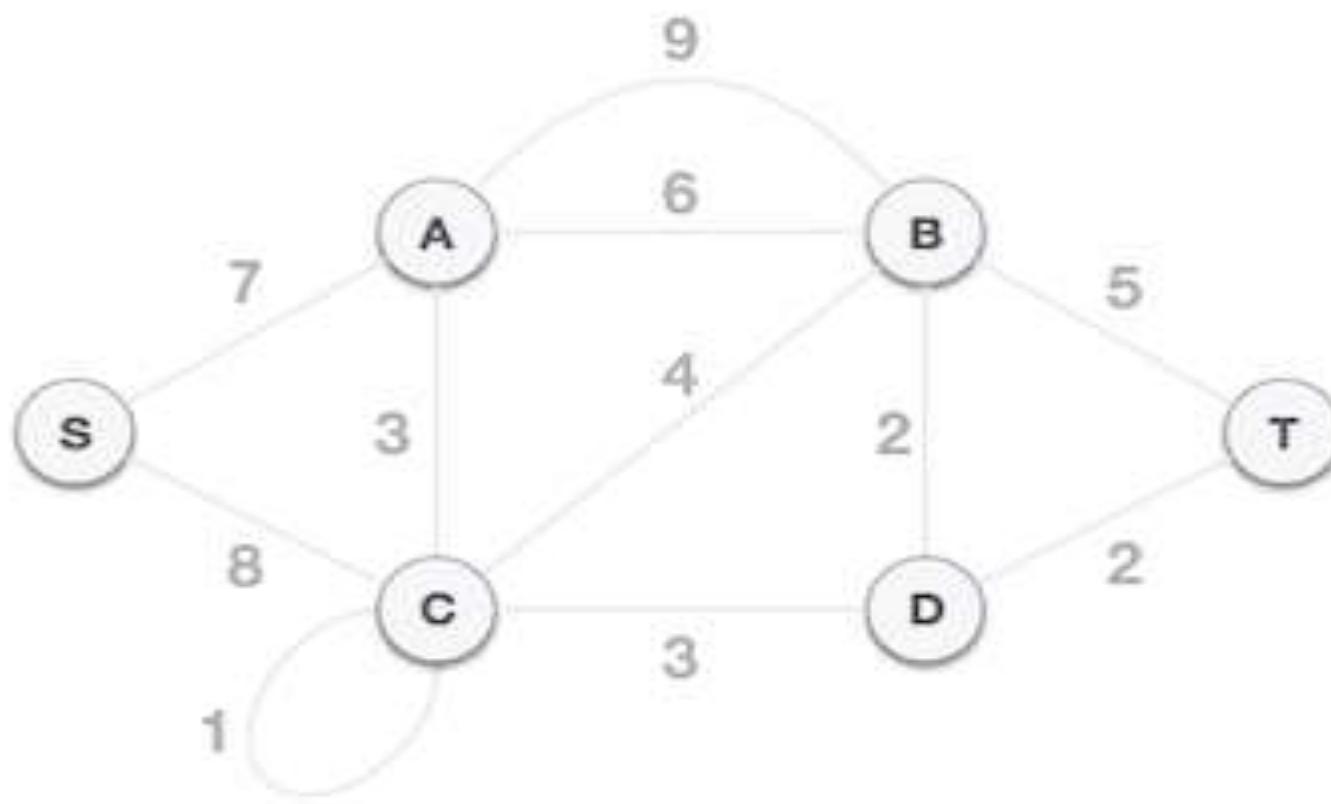
Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- Has the minimum sum of weights among all the trees that can be formed from the graph.

How Kruskal's algorithm works

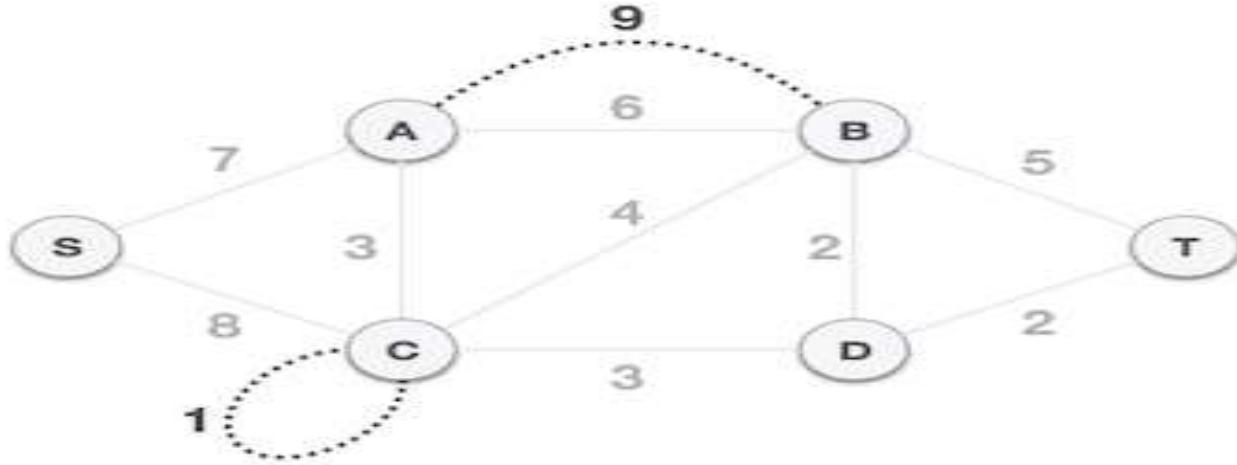
- It falls under a class of algorithms called greedy algorithms which find the local optimum in the hopes of finding a global optimum.
- We start from the edges with the lowest weight and keep adding edges until we reach our goal.
- The steps for implementing Kruskal's algorithm are as follows:
 - Sort all the edges from low weight to high
 - Take the edge with the lowest weight and add it to the spanning tree.
 - If adding the edge created a cycle, then reject this edge.
 - Keep adding edges until we reach all vertices.

Kruskal's Spanning Tree Algorithm

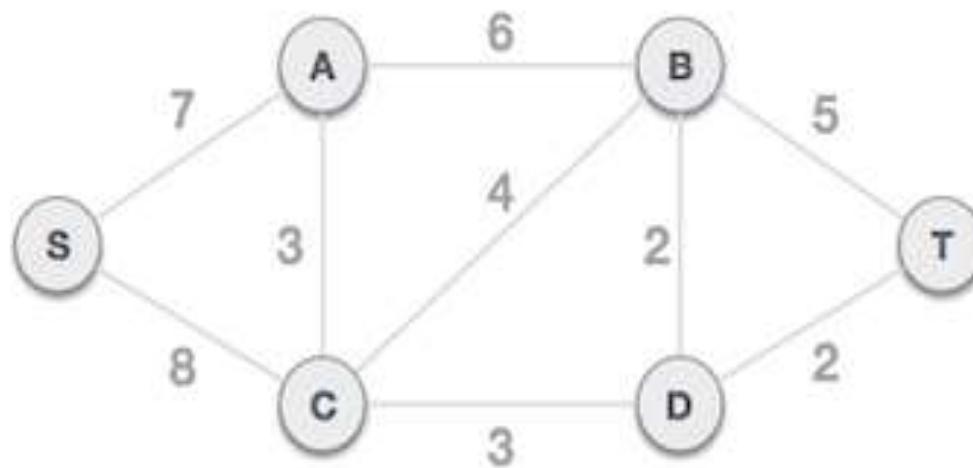


➤ Step 1: Remove all loops and Parallel Edges

➤ Remove all loops and parallel edges from the given graph.



➤ In case of parallel edges, keep the one which has the least cost associated and remove all others.

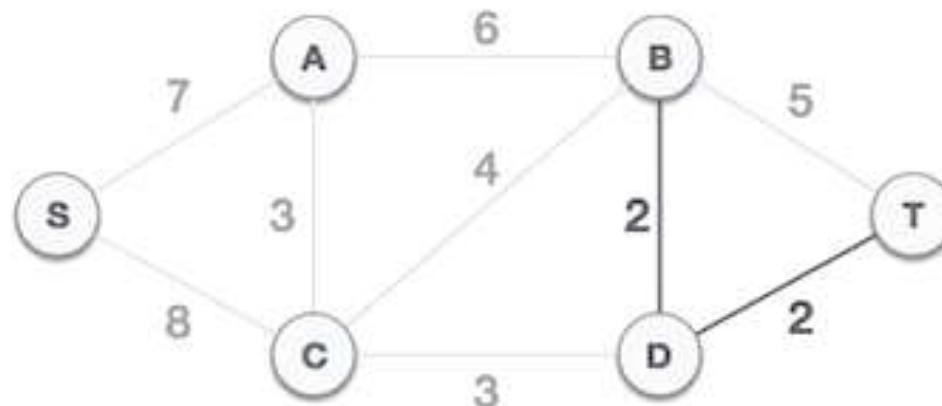


- Step 2 :Arrange all edges in their increasing order of weight
- The next step is to create a set of edges and weight, and arrange them in an ascending order of weight age (cost).

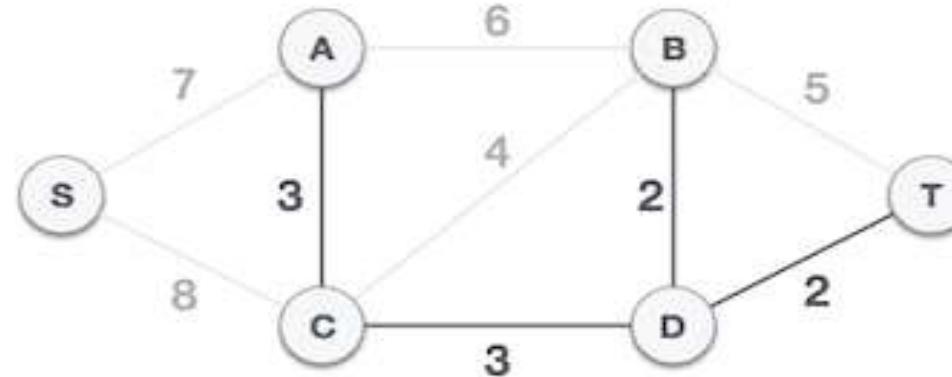
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3 : Add the edge which has the least weight age

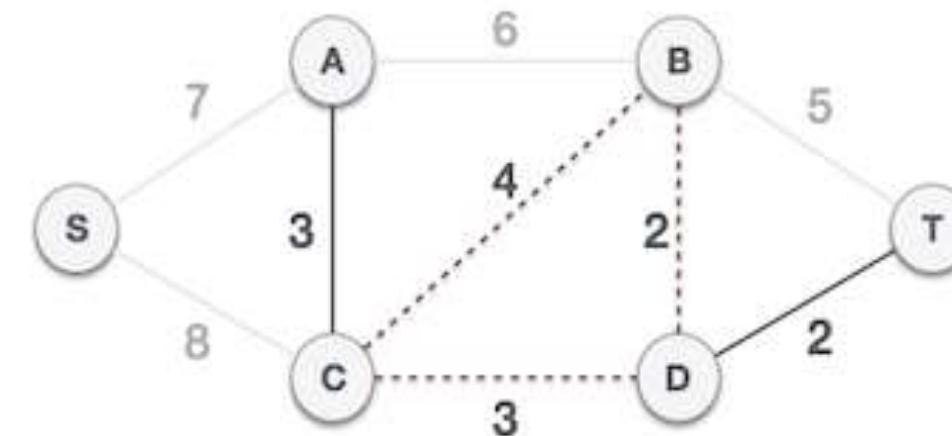
Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.



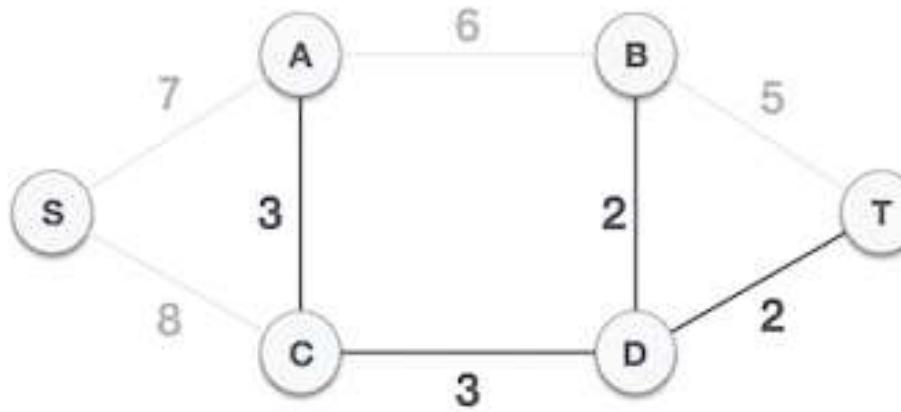
- The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.
- Next cost is 3, and associated edges are A,C and C,D. We add them again



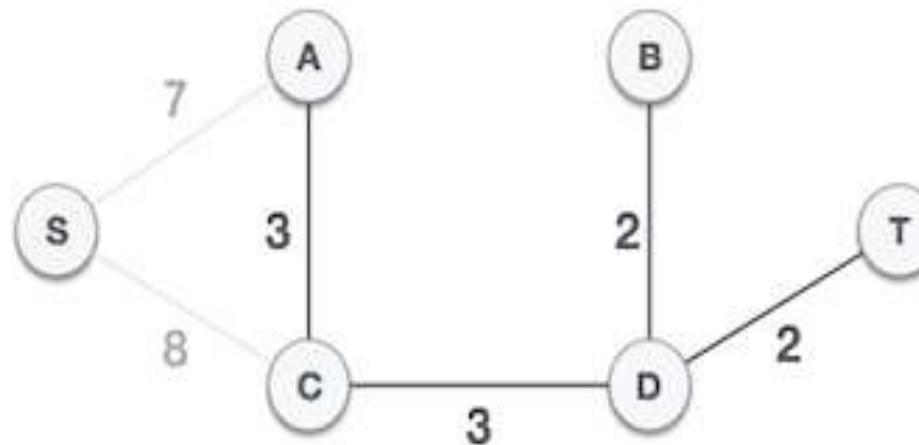
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph.



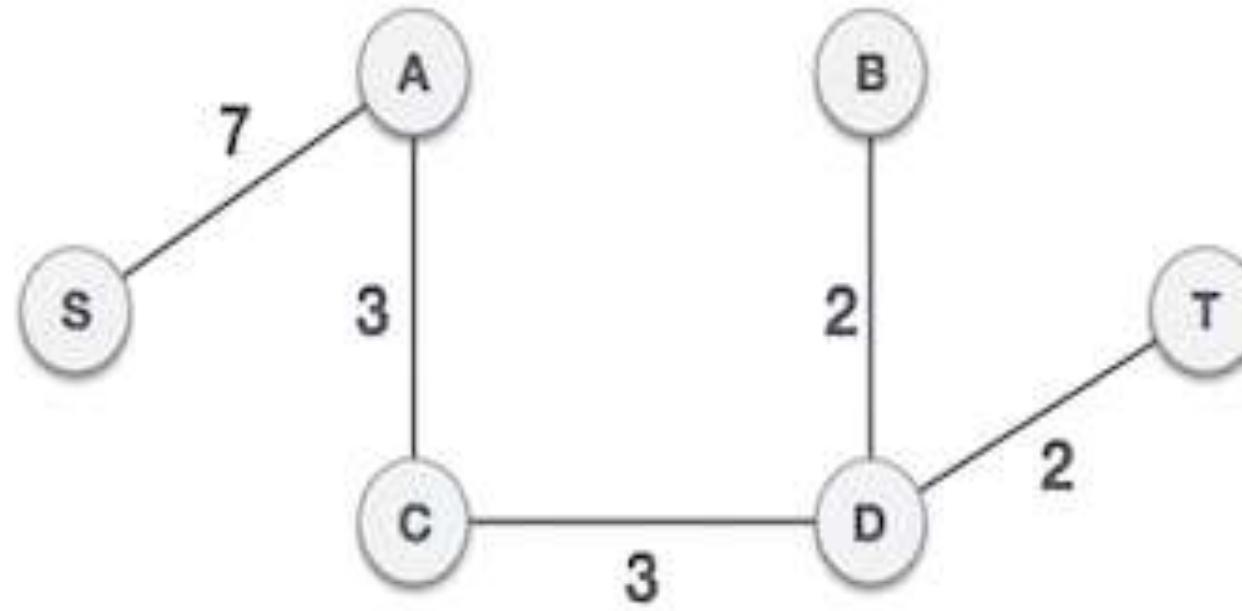
- We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



- Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.

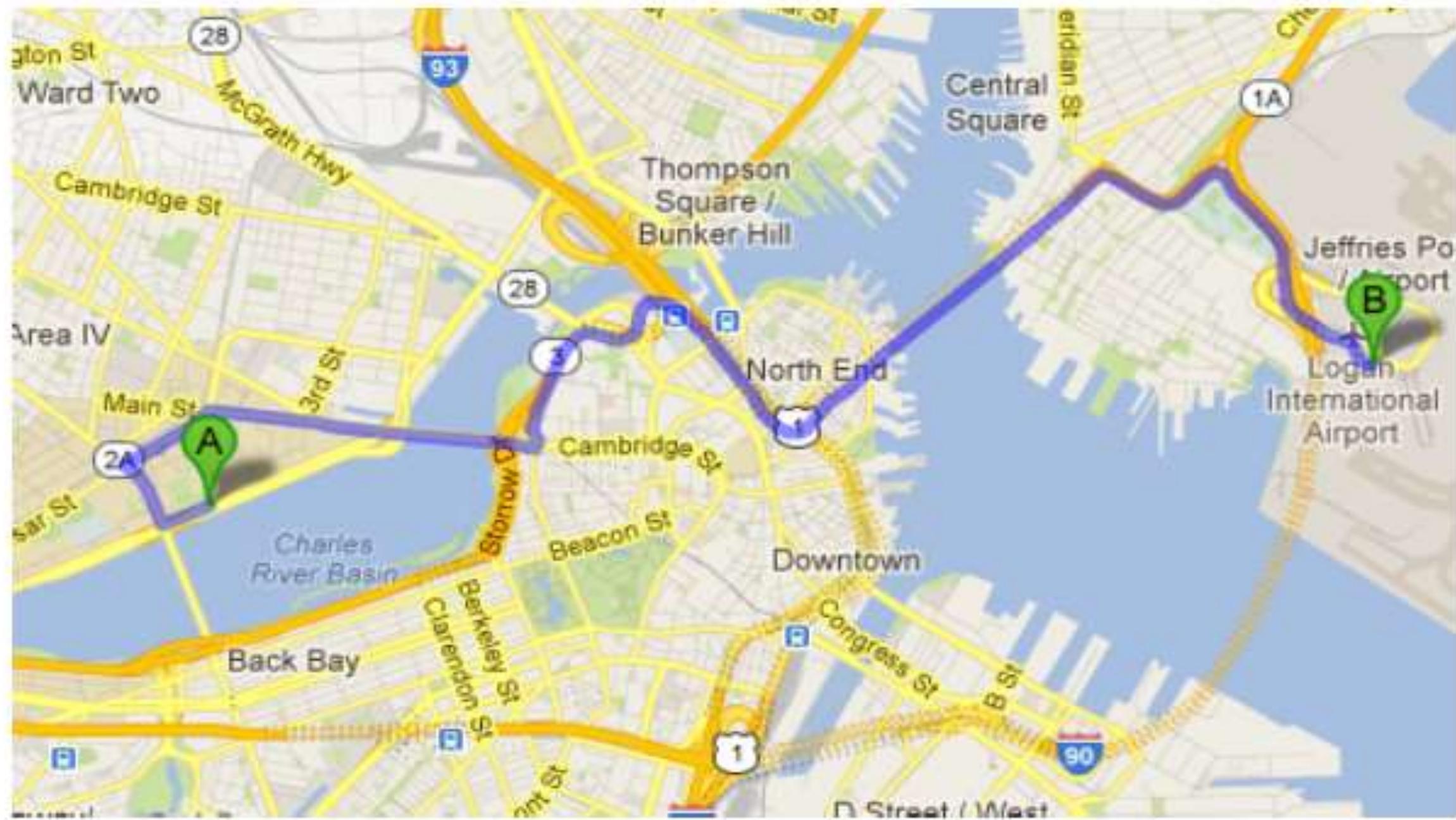


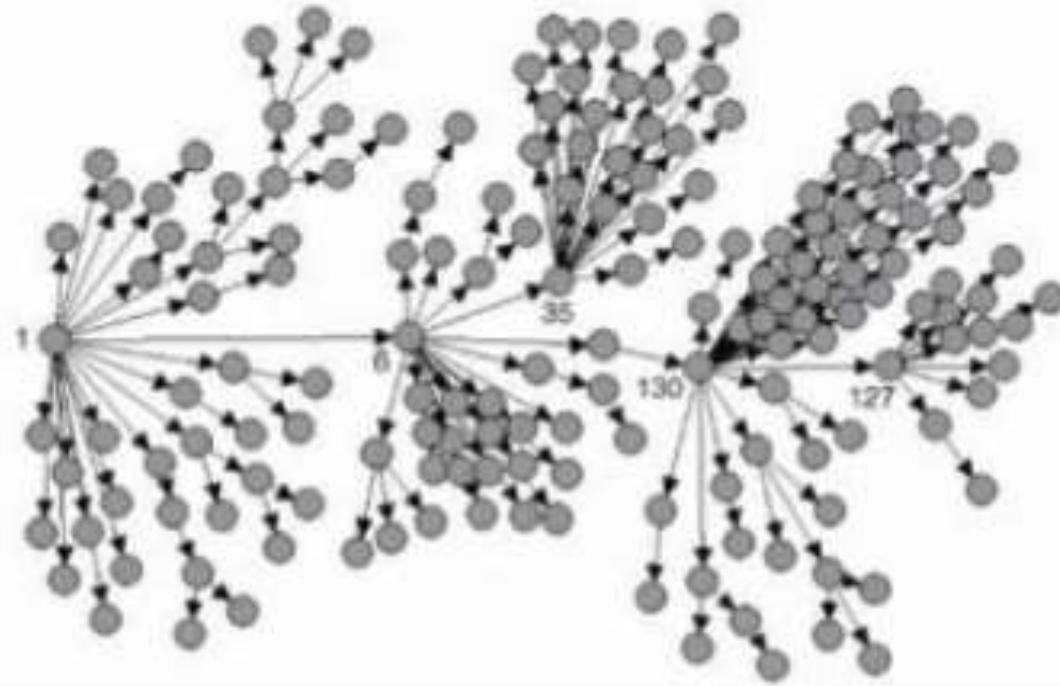
By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Transitive Closure

Dijkstra's Algorithm

- Dijkstra's algorithm, given by a Dutch scientist Edsger Dijkstra in 1959, is used to find the shortest path tree.
- This algorithm is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).
- Given a graph G and a source node A , the algorithm is used to find the shortest path (one having the lowest cost) between A (source node) and every other node.
- Moreover, Dijkstra's algorithm is also used for finding costs of shortest paths from a source node to a destination node.





Simplicity is prerequisite for reliability.
(Edsger Dijkstra)

Dijkstra's Algorithm

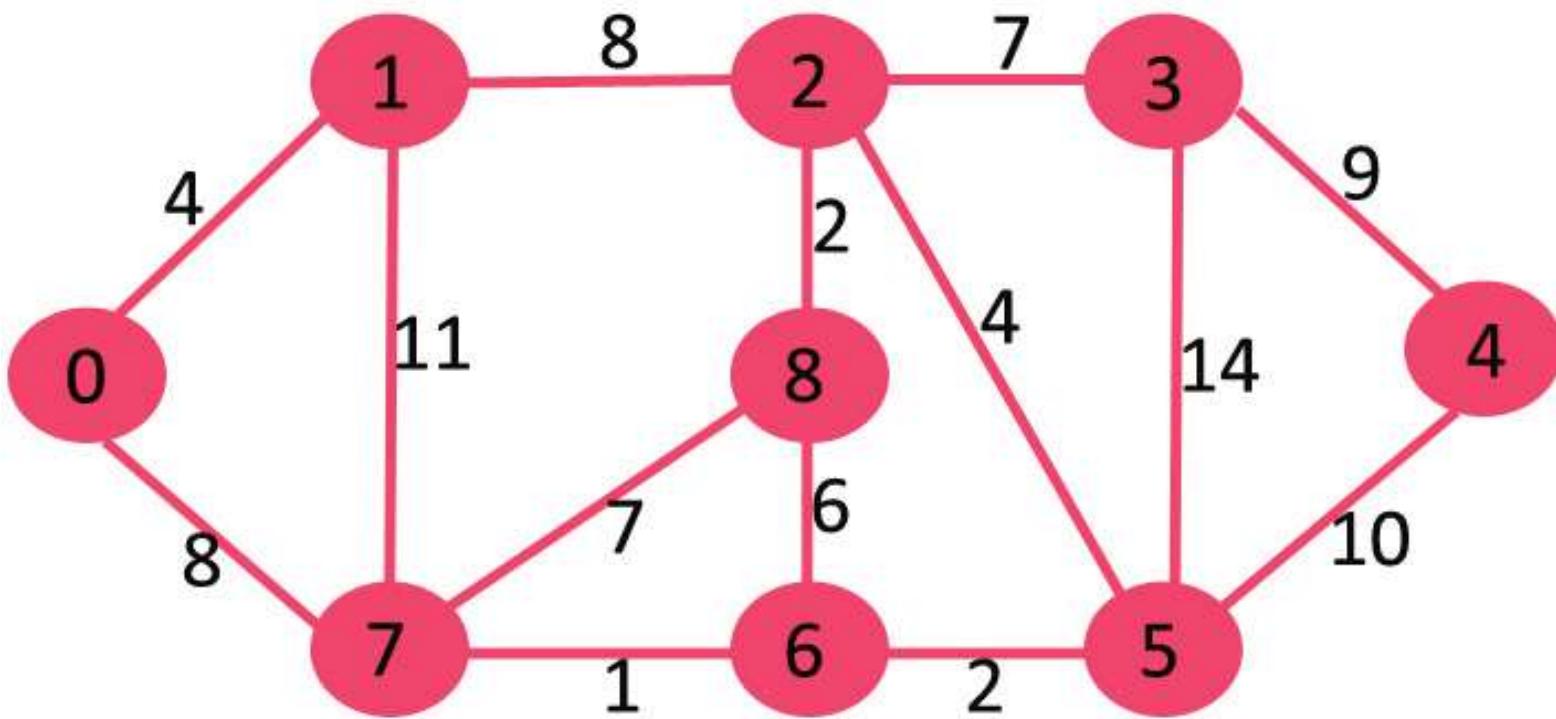
- 1 Select the source node also called the initial node
- 2 Define an empty set N that will be used to hold nodes to which a shortest path has been found.
- 3 Label the initial node with 0, and insert it into N.
- 4 Repeat Steps 5 to 7 until the destination node is in N or there are no more labeled nodes in N.
- 5 Consider each node that is not in N and is connected by an edge from the newly inserted node.
- 6
 - a. If the node that is not in N has no labeled then SET the label of the node = the label of the newly inserted node + the length of the edge.
 - b. Else if the node that is not in N was already labeled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)
- 7 Pick a node not in N that has the smallest label assigned to it and add it to N

Dijkstra's Algorithm

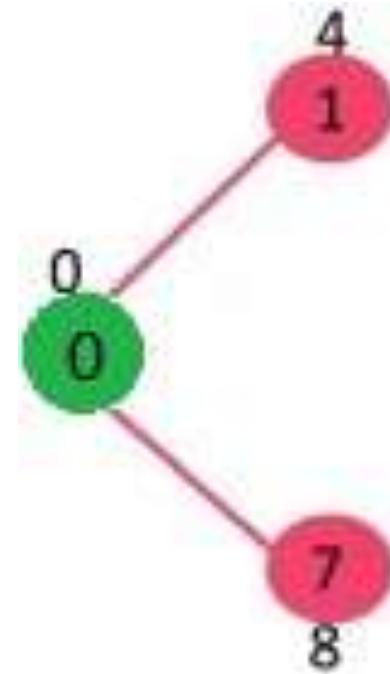
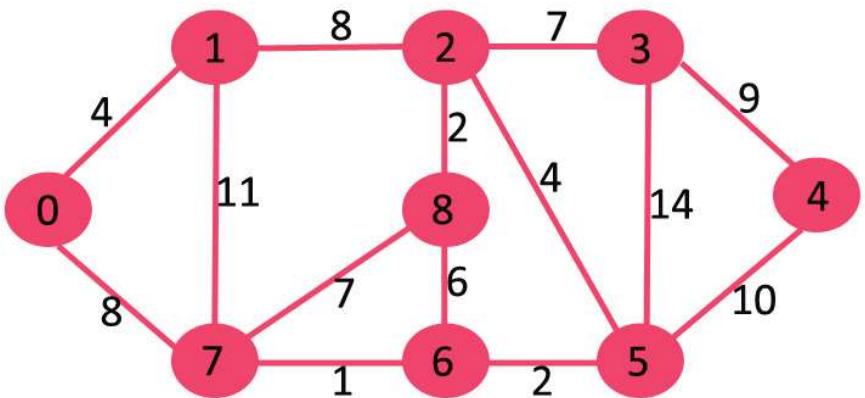
- Dijkstra's algorithm labels every node in the graph where the labels represent the distance (cost) from the source node to that node. There are two kinds of labels: temporary and permanent.
- While temporary labels are assigned to nodes that have not been reached, permanent labels are given to nodes that have been reached and their distance (cost) to the source node is known. A node must be a permanent label or a temporary label, but not both.
- The execution of this algorithm will produce either of two results
- If the destination node is labeled, then the label will in turn represent the distance from the source node to the destination node.
- If the destination node is not labeled, then it means that there is no path from the source to the destination node.

Dijkstra's Example

- Example: Consider the graph G given below. Taking the initial node, execute the Dijkstra's algorithm on it.

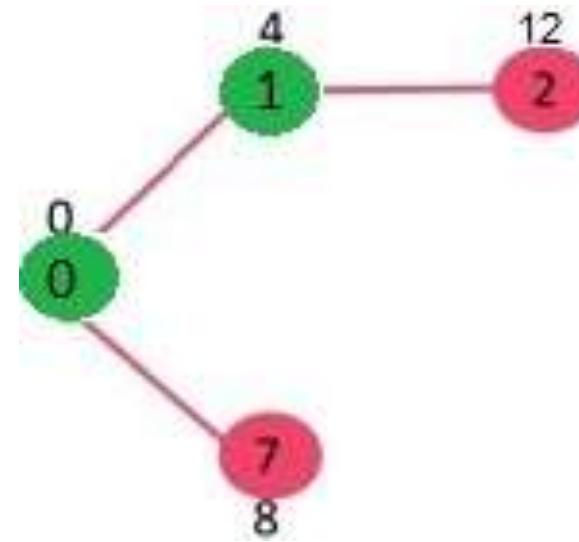
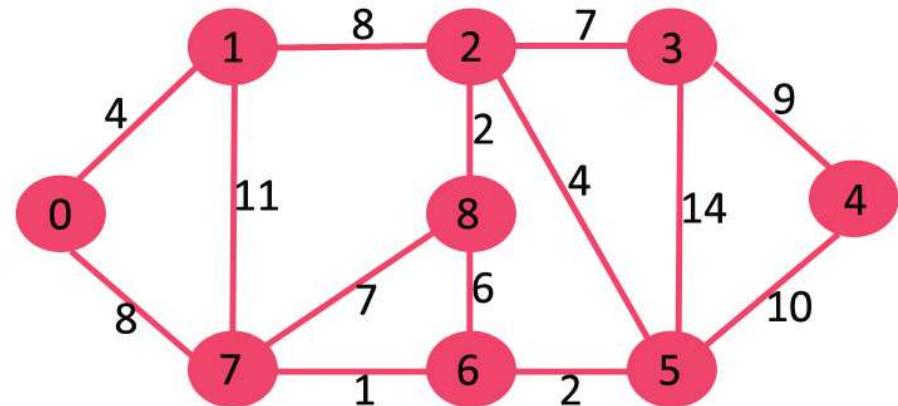


- Adjacent vertices of 0 ,1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

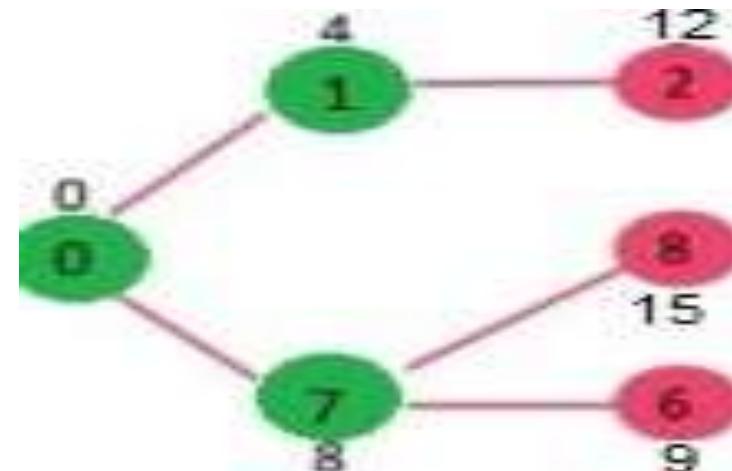
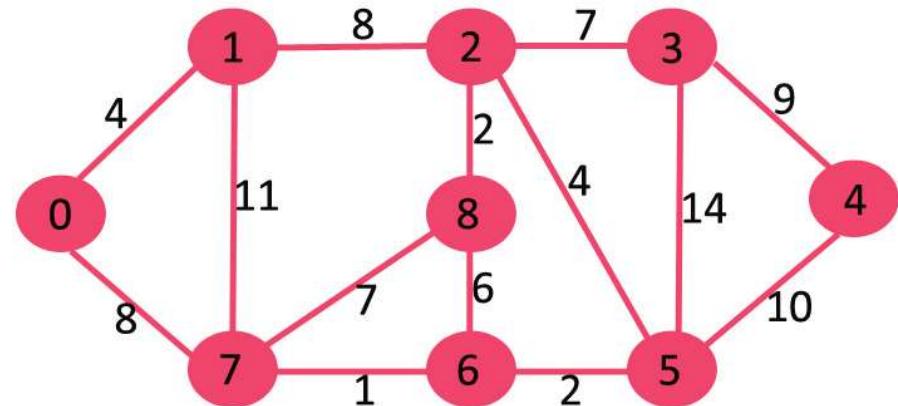


- Following sub graph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.

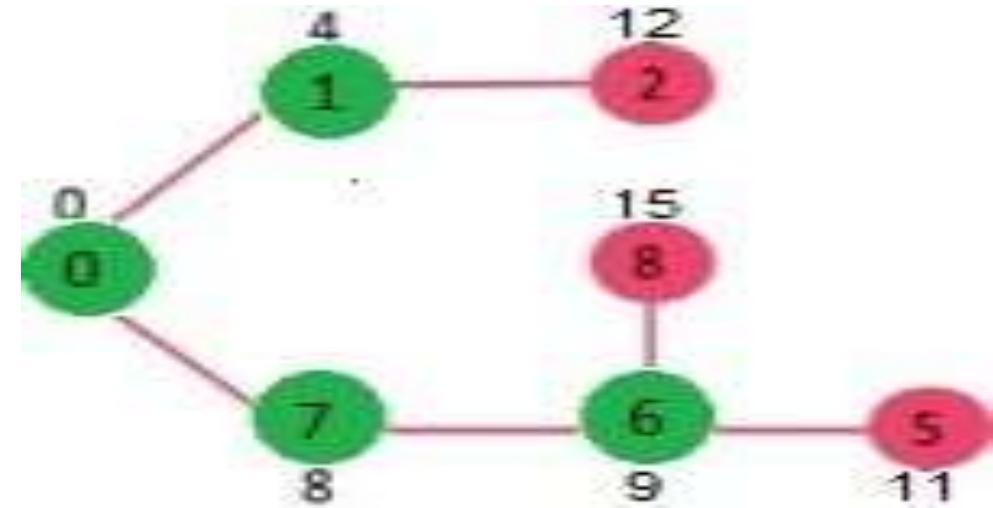
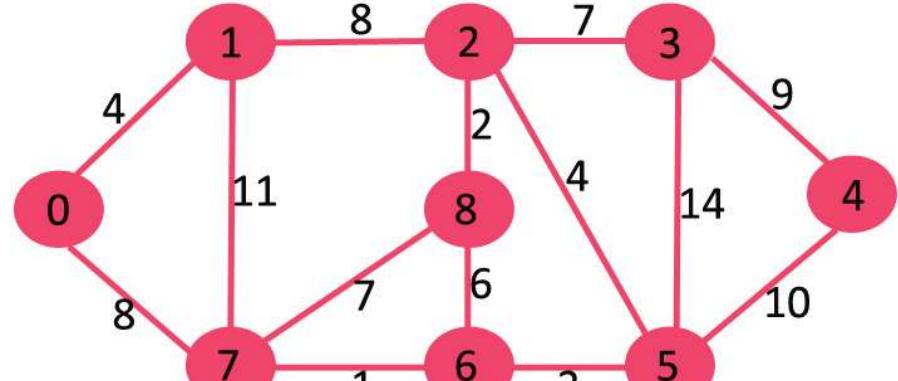
The vertex 1 is picked and added to SPT Set. So SPT Set now becomes $\{0, 1\}$. Update the distance values of adjacent vertices of 1. The distance value of vertex 2



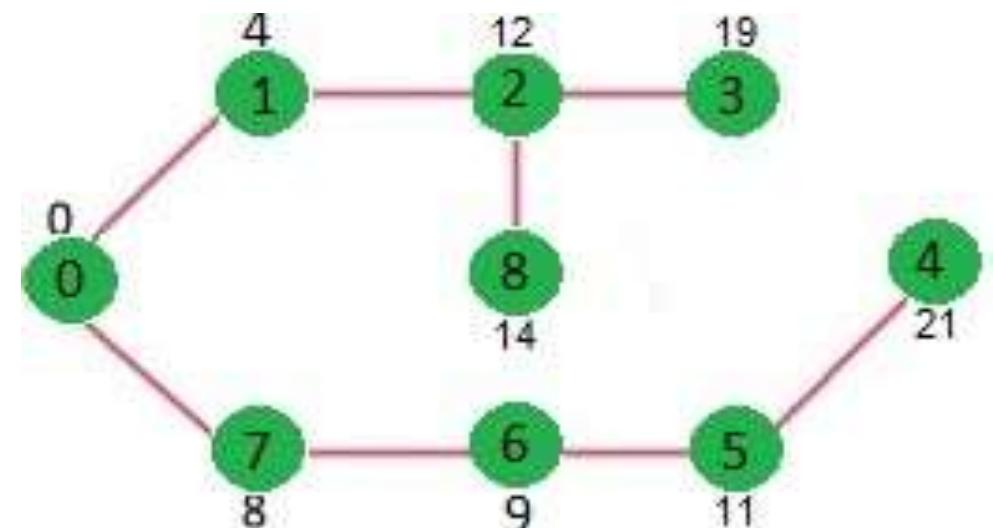
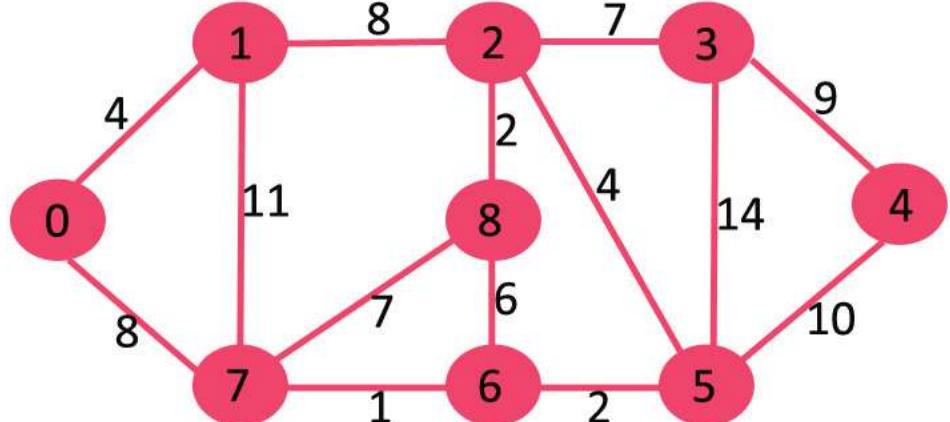
Pick the vertex with minimum distance value and not already included in SPT (not in SPT SET). Vertex 7 is picked. So SPT Set now becomes $\{0, 1, 7\}$. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in SPT SET). Vertex 6 is picked. So SPT Set now becomes $\{0, 1, 7, 6\}$. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until SPT SET does include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



War shall Algorithm

WARSHALL ALGORITHM

- Floyd Warshall Algorithm is a famous algorithm.
- It is used to solve All Pairs Shortest Path Problem.
- It computes the shortest path between every pair of vertices of the given graph.
- Floyd Warshall Algorithm is an example of dynamic programming approach.

ADVANTAGES

- It is extremely simple.
- It is easy to implement.

TIME COMPLEXITY

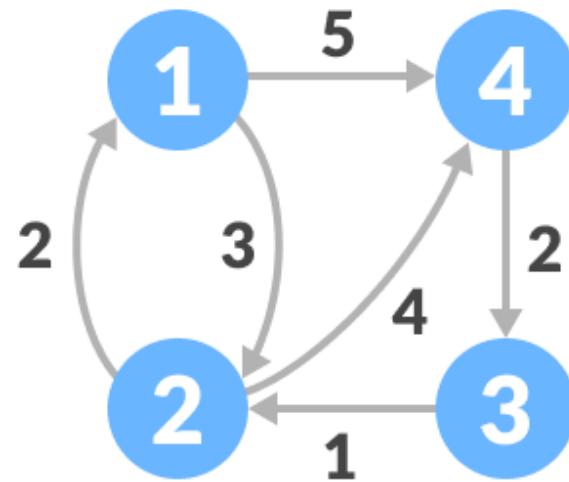
- Floyd Warshall Algorithm consists of three loops over all the nodes.
- The inner most loop consists of only constant complexity operations.
- Hence, the asymptotic complexity of Floyd Warshall algorithm is $O(n^3)$.
- Here, n is the number of nodes in the given graph.

WHEN FLOYD WARSHALL ALGORITHM IS USED?

- Floyd Warshall Algorithm is best suited for dense graphs.
- This is because its complexity depends only on the number of vertices in the given graph.
- For sparse graphs, Johnson's Algorithm is more suitable.

PROBLEM BASED ON FLOYD WARSHALL ALGORITHM

Consider the following directed weighted graph



Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.

STEP-01:

- Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self edges nor parallel edges.

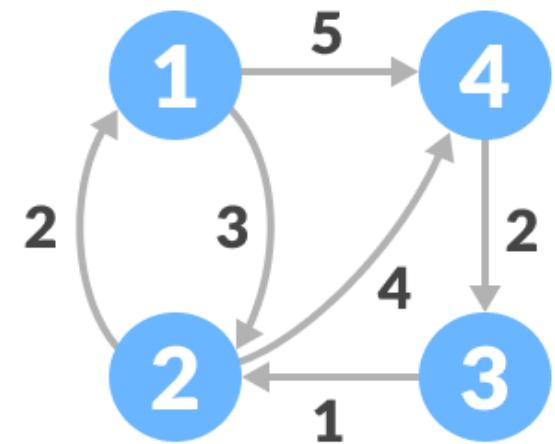
STEP-02:

Write the initial distance matrix.

- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value = ∞ .

Initial distance matrix for the given graph is

$$A^0 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & \infty & 1 & 0 & \infty \\ 4 & \infty & \infty & 2 & 0 \end{bmatrix}$$



STEP-03:

➤ Using Floyd Warshall Algorithm, write the following 4 matrices

$$A^1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & 0 & 0 \\ 3 & \infty & 0 & 0 & 0 \\ 4 & \infty & 0 & 0 & 0 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & \infty & 1 & 0 & 8 \\ 4 & \infty & \infty & 2 & 0 \end{bmatrix}$$

- In a similar way, A_2 is created using A_3 . The elements in the second column and the second row are left as they are.
- In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in step 3.

FROM THE A_1 →

$$A_3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & \infty & 1 & 0 & 8 \\ 4 & \infty & \infty & 2 & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & & \\ 2 & 2 & 0 & 9 & 4 \\ 3 & & 1 & 0 & \\ 4 & & \infty & 0 & \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & \infty & \infty & 2 & 0 \end{bmatrix}$$

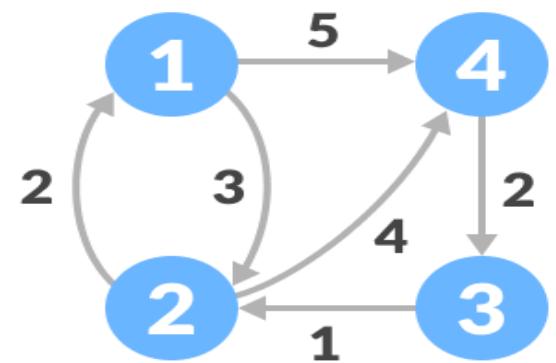
SIMILARLY, A3 AND A4 IS ALSO CREATED.

FROM THE A2

$$\xrightarrow{\hspace{1cm}} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & \infty & \infty & 2 & 0 \end{bmatrix} \xrightarrow{\hspace{1cm}} \boxed{\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ 3 & \infty & 1 & 0 & 8 \\ 4 & 2 & 0 & 0 \end{bmatrix}}$$

$$A^3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ 3 & \infty & 1 & 0 & 8 \\ 4 & 2 & 0 & 0 \end{bmatrix} \xrightarrow{\hspace{1cm}} \boxed{\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{bmatrix}}$$

$$A^4 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 5 \\ 2 & 0 & 4 \\ 3 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{bmatrix} \xrightarrow{\hspace{1cm}} \boxed{\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{bmatrix}}$$



Floyd Warshall Algorithm Applications

- To find the shortest path in a directed graph
- To find the transitive closure of directed graphs
- To find the Inversion of real matrices
- For testing whether an undirected graph is bipartite

// Floyd-Warshall Algorithm in C

```
#include <stdio.h>
// defining the number of vertices
#define nV 4
#define INF 999
void printMatrix(int A[][nV]);
void floydWarshall(int graph[][nV])
{
    int A[nV][nV], i, j, k;
    for (i = 0; i < nV; i++)
        for (j = 0; j < nV; j++)
            A[i][j] = graph[i][j];
    for (k = 0; k < nV; k++)
    {
        for (i = 0; i < nV; i++)
        {
            for (j = 0; j < nV; j++)
            {
                if (A[i][k] + A[k][j] < A[i][j])
                    A[i][j] = A[i][k] + A[k][j];
            }
        }
    }
}
```

```
}
```

```
}
```

```
}
```

```
printMatrix(A);
}
```

```
void printMatrix(int A[][nV])
{
    for (int i = 0; i < nV; i++)
    {
        for (int j = 0; j < nV; j++)
        {
            if (A[i][j] == INF)
                printf("%4s", "INF");
            else
                printf("%4d", A[i][j]);
        }
        printf("\n");
    }
}
```

```
int main()
{
    int graph[nV][nV] = {{0, 3, INF, 5},
                          {2, 0, INF, 4},
                          {INF, 1, 0, INF},
                          {INF, INF, 2, 0}};
    floydWarshall(graph);
}
```

TOPOLOGICAL SORT

TOPOLOGICAL SORT

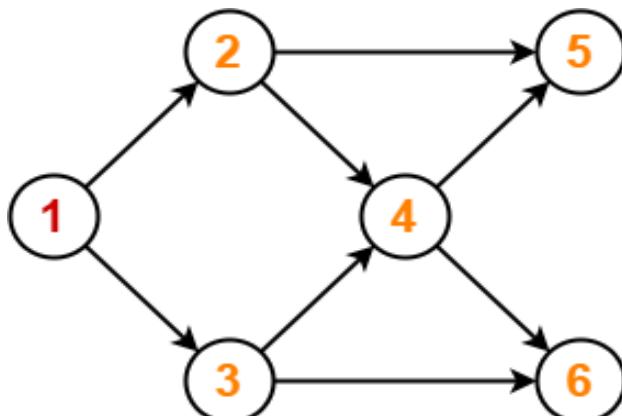
Topological Sort is a linear ordering of the vertices in such a way that if there is an edge in the DAG going from vertex ‘u’ to vertex ‘v’, then ‘u’ comes before ‘v’ in the ordering.

- Topological Sorting is possible if and only if the graph is a Directed Acyclic Graph.
- There may exist multiple different topological orderings for a given directed acyclic graph.
- In simple words, a topological ordering of a DAG G, is an ordering of its vertices such that any directed path in G, traverses vertices in increasing order.

Applications of Topological Sort

Few important applications of topological sort are

- Scheduling jobs from the given dependencies among jobs
- Instruction Scheduling
- Determining the order of compilation tasks to perform in makefiles
- Data Serialization



Topological Sort Example

For this graph, following 4 different topological orderings are possible

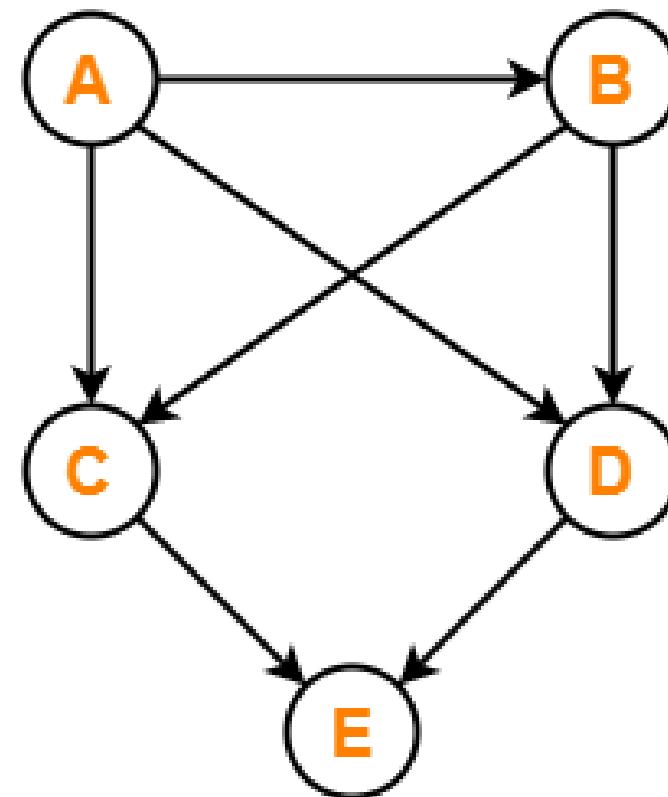
- 1 2 3 4 5 6
- 1 2 3 4 6 5
- 1 3 2 4 5 6
- 1 3 2 4 6 5

Topological Sorting

- Algorithm to find a Topological Sort T of a Directed Acyclic Graph, G
- Step 1: Find the indegree $\text{INDEG}(N)$ of every node in the graph
- Step 2: Enqueue all the nodes with a zero in-degree
- Step 3: Repeat Steps 4 and 5 until the QUEUE is empty
- Step 4: Remove the front node N of the QUEUE by setting $\text{FRONT} = \text{FRONT} + 1$
- Step 5: Repeat for each neighbor M of node N:
 - a) Delete the edge from N to M by setting $\text{INDEG}(M) = \text{INDEG}(M) - 1$
 - b) IF $\text{INDEG}(M) = 0$, then Enqueue M, that is, add M to the rear of the queue
- [END OF INNER LOOP]
- [END OF LOOP]
- Step 6: Exit

PRACTICE PROBLEMS BASED ON TOPOLOGICAL SORT

Find the number of different topological orderings possible for the given graph

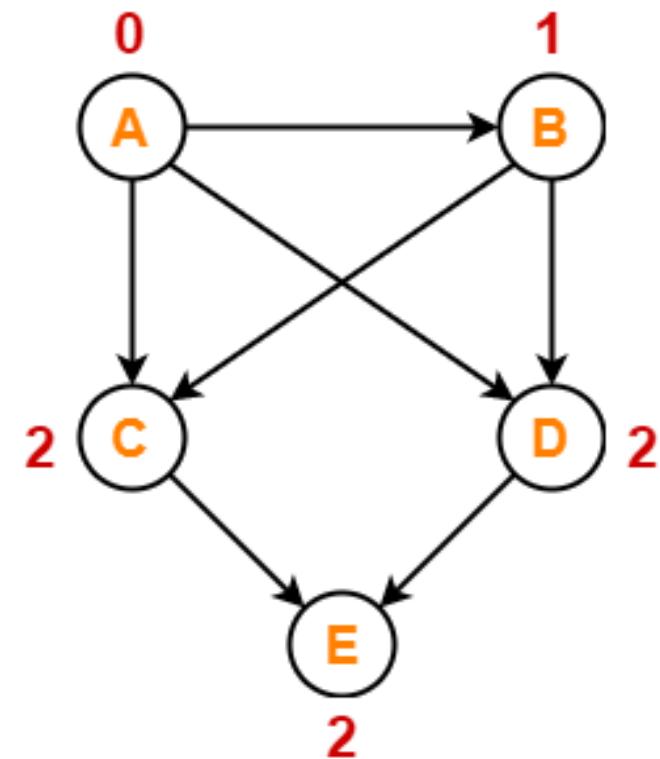
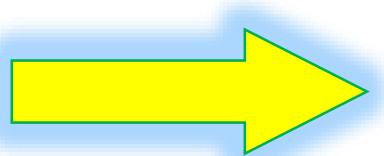
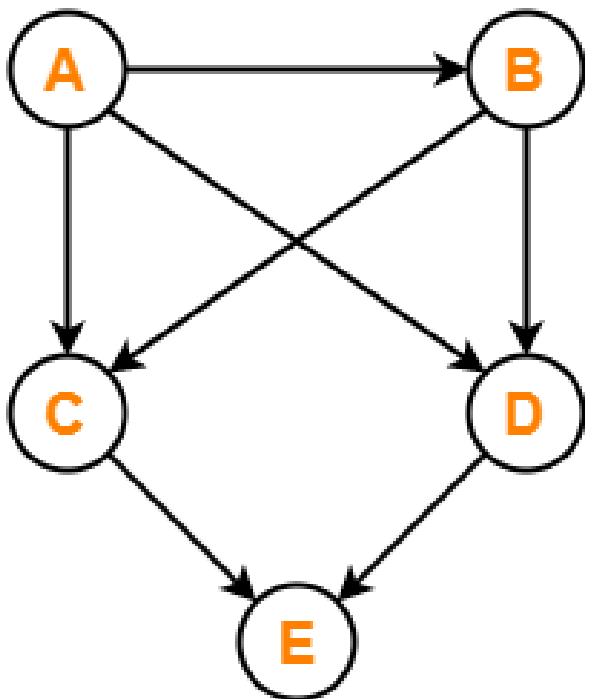


SOLUTION

The topological orderings of the above graph are found in the following steps

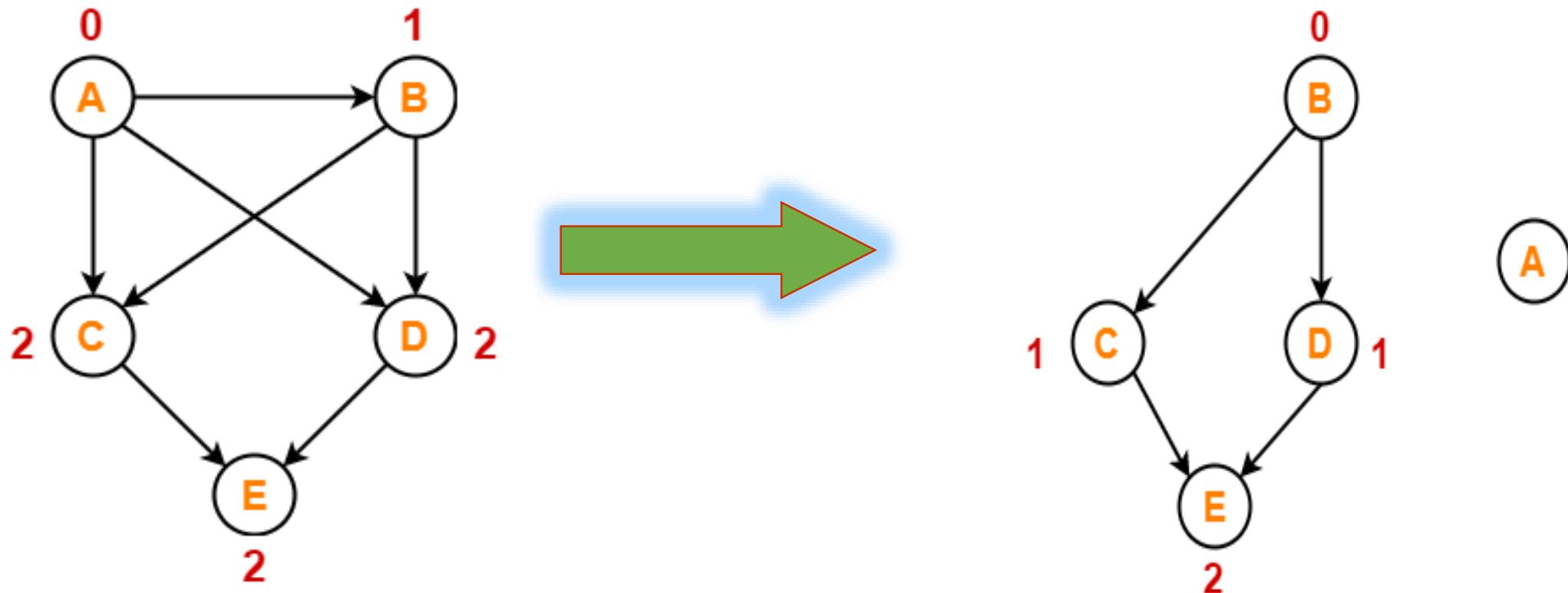
STEP-01:

Write in-degree of each vertex



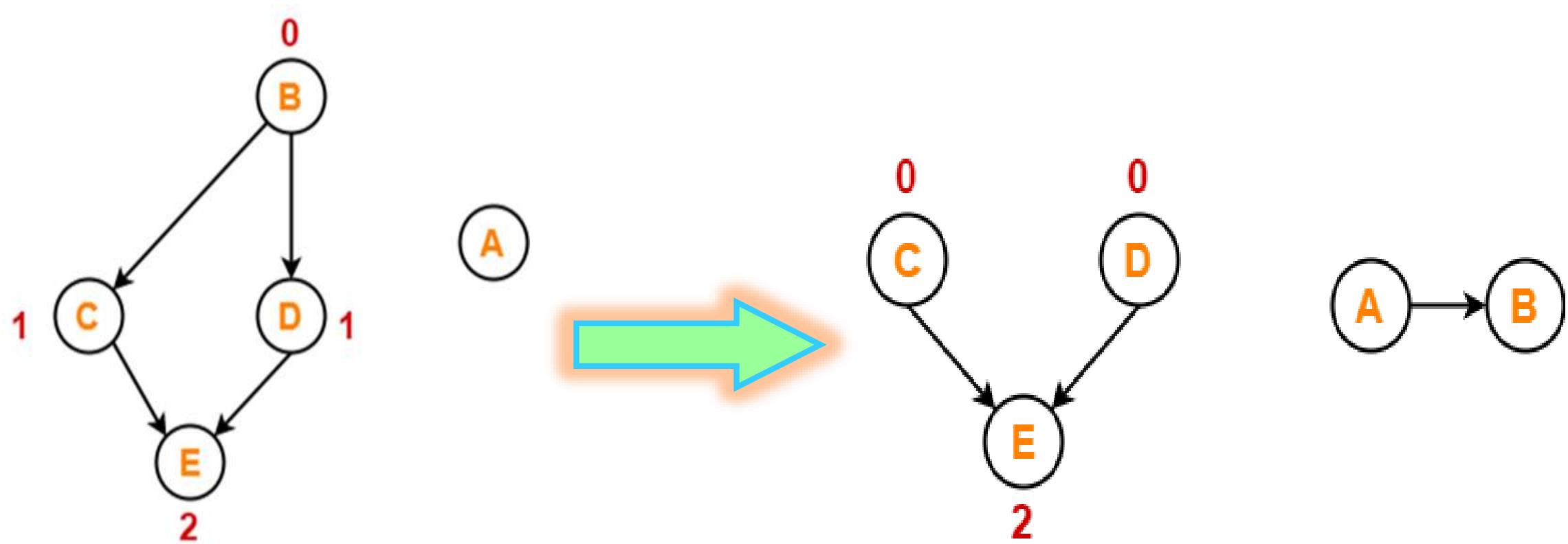
STEP-02:

- Vertex-A has the least in-degree.
- So, remove vertex-A and its associated edges.
- Now, update the in-degree of other vertices.



STEP-03:

- Vertex-B has the least in-degree.
- So, remove vertex-B and its associated edges.
- Now, update the in-degree of other vertices.



STEP-04:

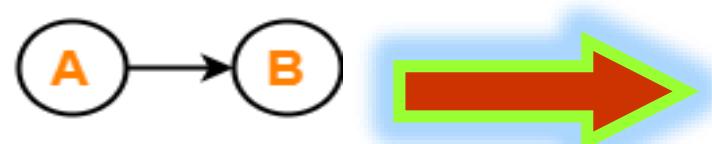
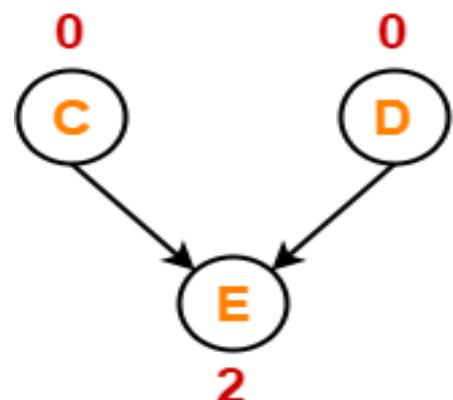
There are two vertices with the least in-degree. So, following 2 cases are possible.

In case-01,

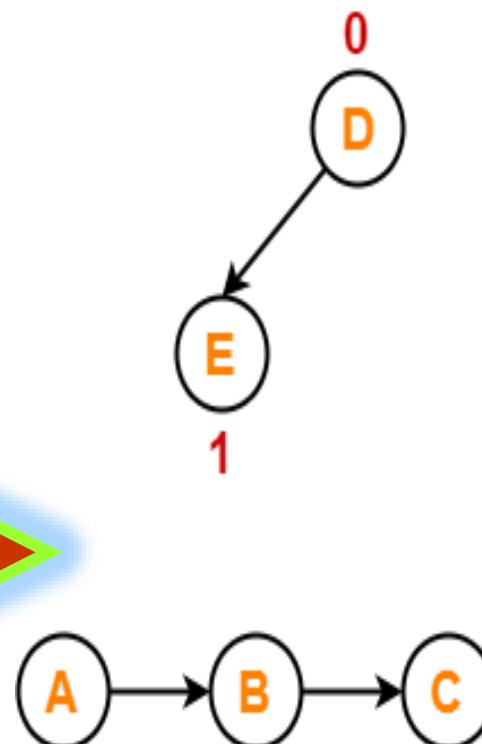
- Remove vertex-C and its associated edges.
- Then, update the in-degree of other vertices.

In case-02,

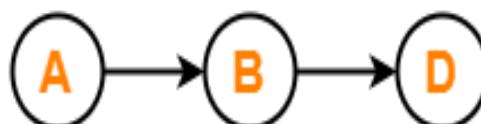
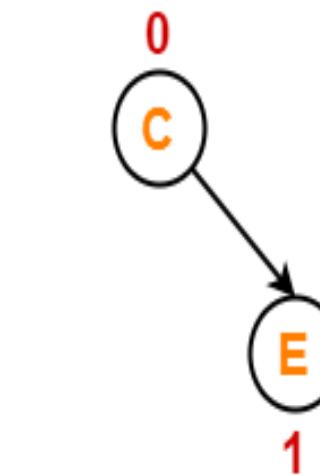
- Remove vertex-D and its associated edges.
- Then, update the in-degree of other vertices.



Case-01



Case-02



STEP-05:

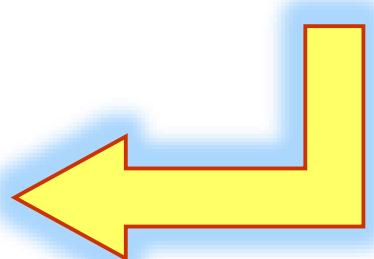
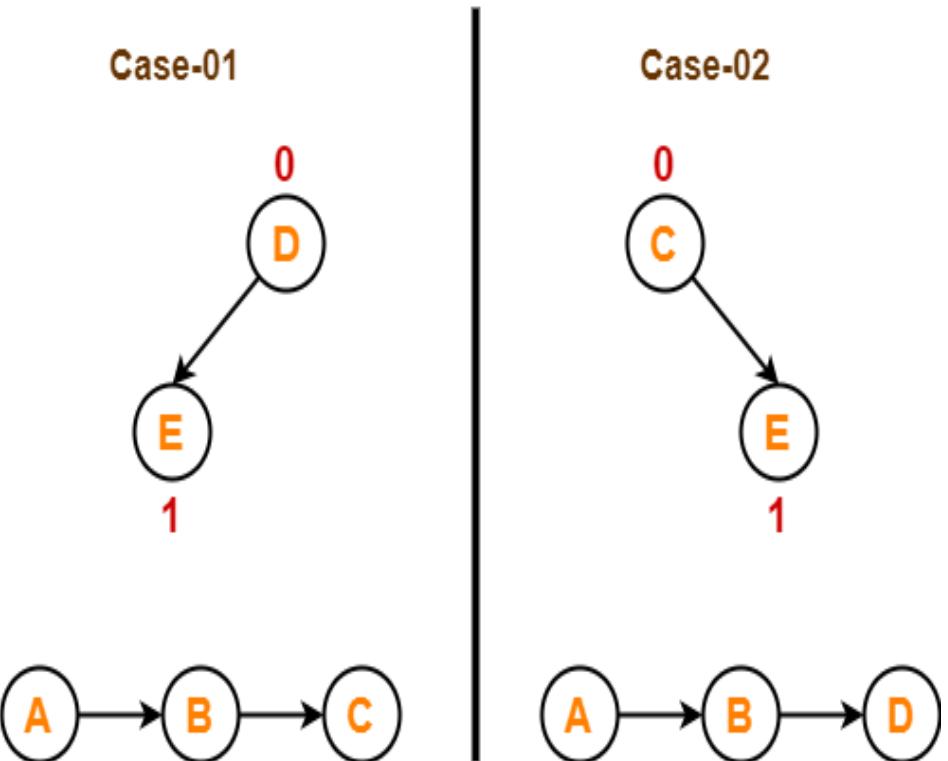
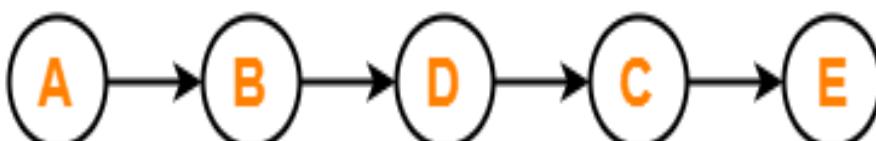
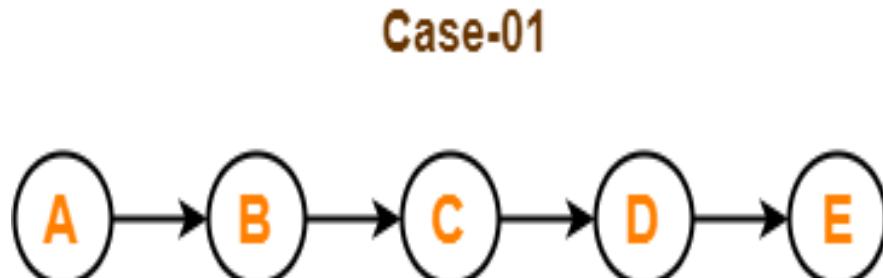
Now, the above two cases are continued separately in the similar manner.

In case-01,

- Remove vertex-D since it has the least in-degree.
- Then, remove the remaining vertex-E.

In case-02,

- Remove vertex-C since it has the least in-degree.
- Then, remove the remaining vertex-E.



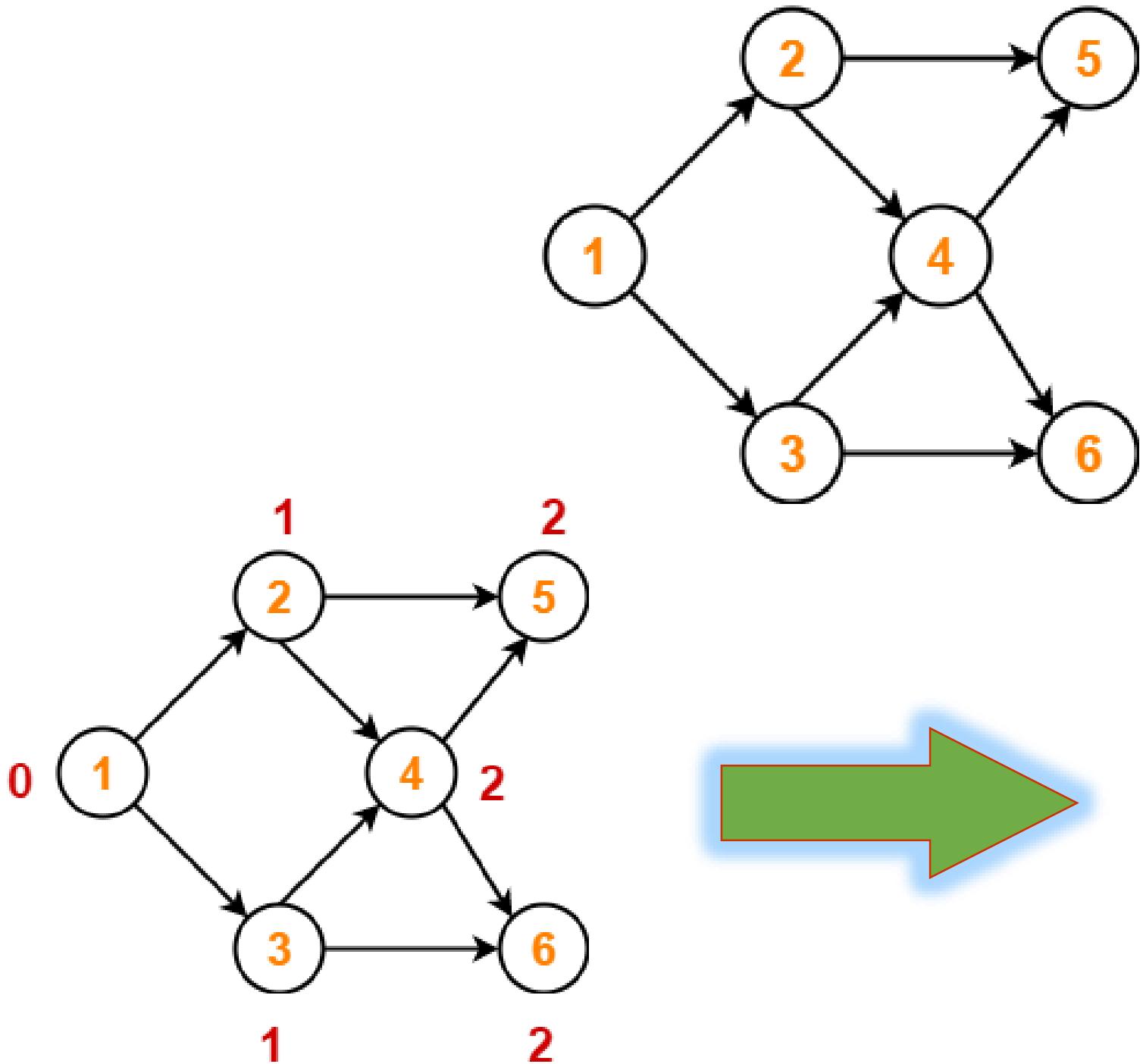
CONCLUSION

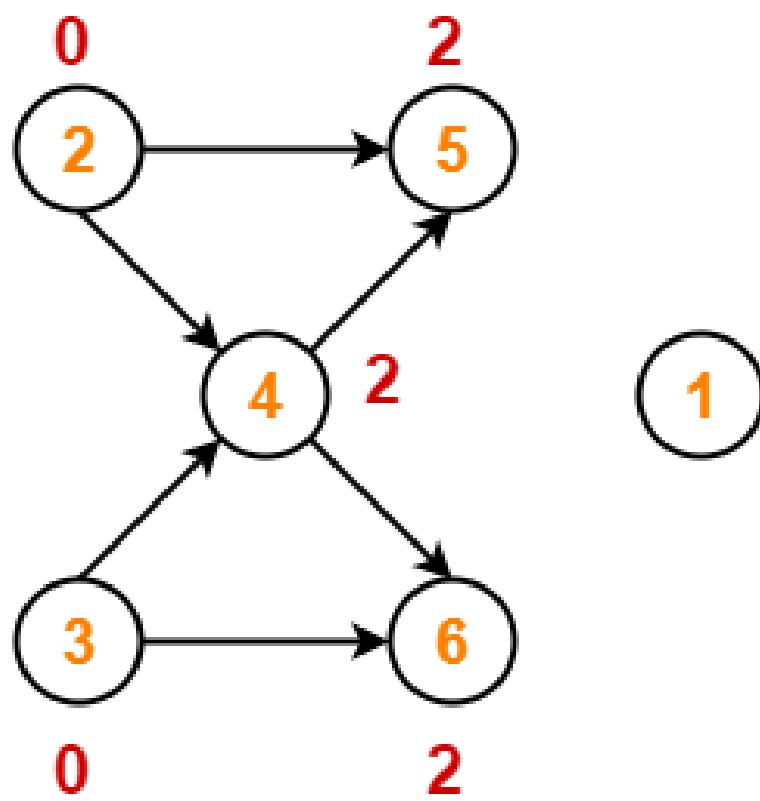
For the given graph, following 2 different topological orderings are possible

- **A B C D E**
- **A B D C E**

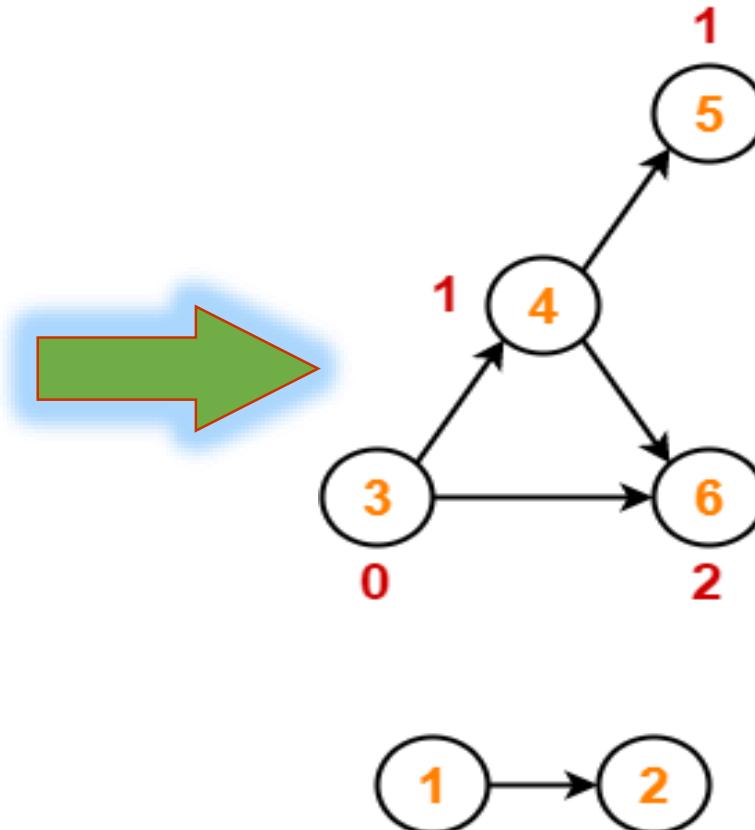
- Topological sorting is widely used in scheduling applications, jobs or tasks. The jobs that have to be completed are represented by nodes, and there is an edge from node u to v if job u must be completed before job v can be started. A topological sort of such a graph gives an order in which the given jobs must be performed.

EXAMPLE - 2

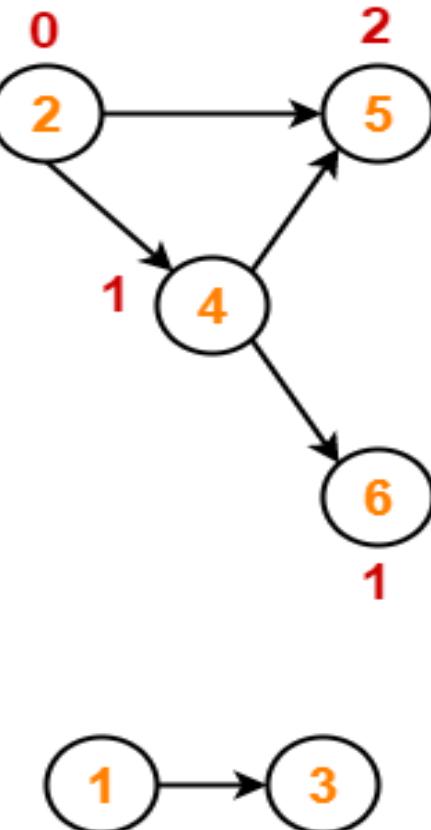




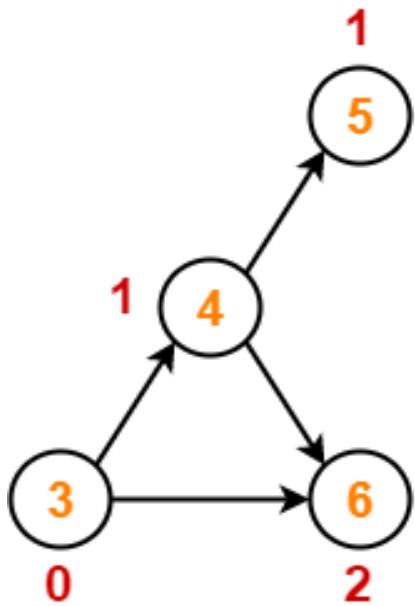
Case-01



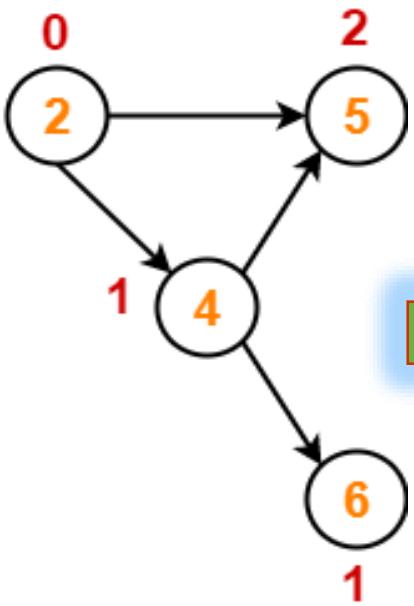
Case-02



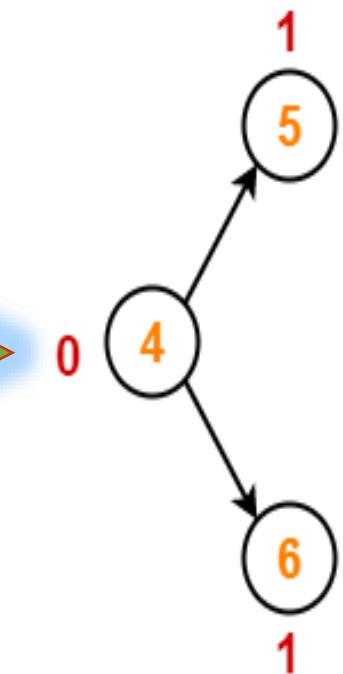
Case-01



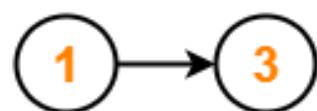
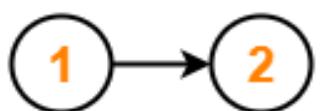
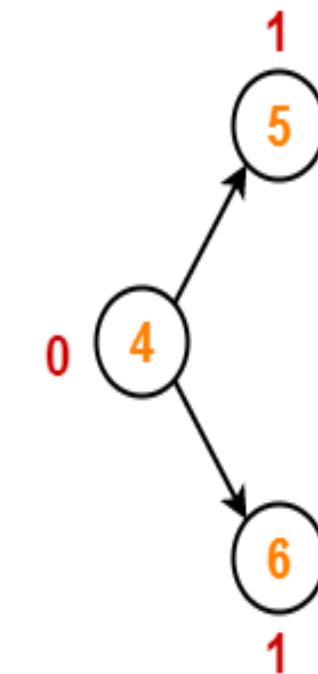
Case-02



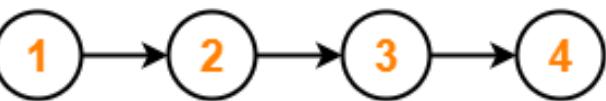
Case-01



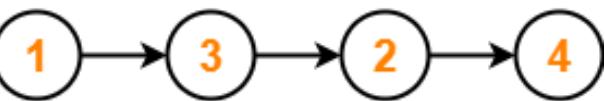
Case-02



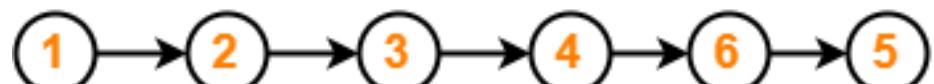
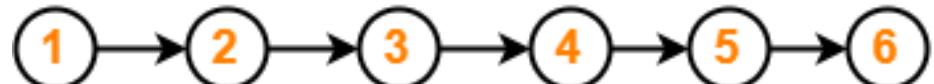
Case-01



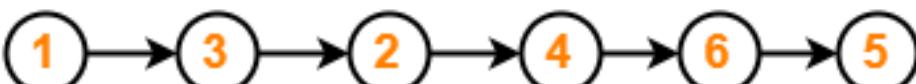
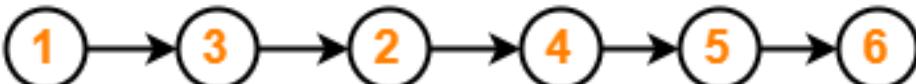
Case-02



From Case-01



From Case-02



➤ 1 2 3 4 5 6

➤ 1 2 3 4 6 5

➤ 1 3 2 4 5 6

➤ 1 3 2 4 6 5

UNIT 4



1. SORTING



Logic &
Sorting



3 5 18 21 24 ...

2 7 12 16 33 ...

2 ...

Sorting

Introduction to Sorting

- ✓ Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.
 - ✓ The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios
-
- ✓ **Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
 - ✓ **Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

Sorting

Sorting Efficiency

- There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters. First parameter is the execution time of program, which means time taken for execution of program. Second is the space, which means space taken by the program.

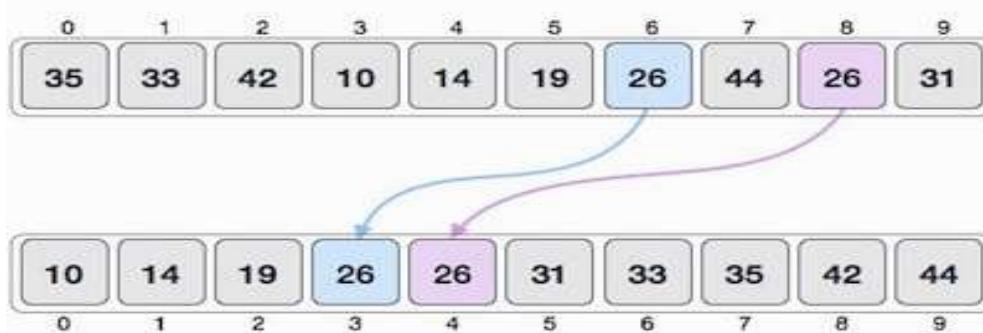
In-place Sorting and Not-in-place Sorting

- Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting**. Bubble sort is an example of in-place sorting.
- However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called **not-in-place sorting**. Merge-sort is an example of not-in-place sorting.

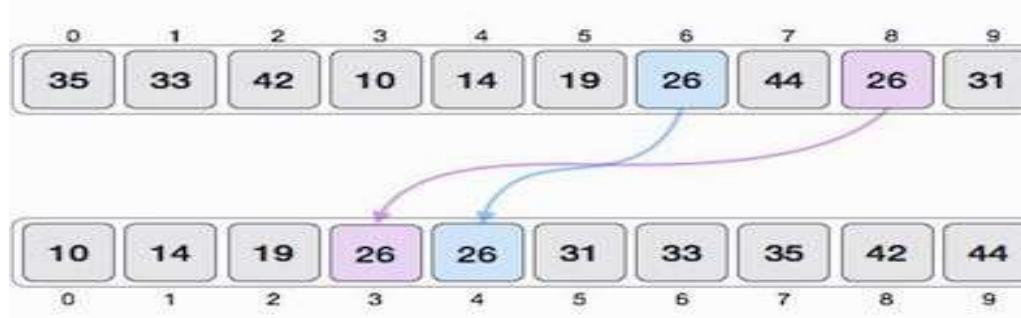
Sorting

✓ Stable and Not Stable Sorting

- ✓ If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.



- ✓ If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



- ✓ Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Sorting

➤ Adaptive and Non-Adaptive Sorting Algorithm

- A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.
- A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

➤ Important Terms

- Some terms are generally coined while discussing sorting techniques, here is a brief introduction to them

➤ Increasing Order

- A sequence of values is said to be in **increasing order**, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.

Sorting

➤ Decreasing Order

- A sequence of values is said to be in **decreasing order**, if the successive element is less than the current one. **For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.**

➤ Non-Increasing Order

- A sequence of values is said to be in **non-increasing order**, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. **For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.**

➤ Non-Decreasing Order

- A sequence of values is said to be in **non-decreasing order**, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. **For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.**

Sorting

➤ Types of Sorting Techniques

- ✓ Bubble Sort
- ✓ Insertion Sort
- ✓ Selection Sort
- ✓ Merge Sort
- ✓ Shell sort
- ✓ Quick Sort
- ✓ Heap Sort
- ✓ Radix sort

BUBBLE -SORT ALGORITHM

6 5 3 1 8 7 2 4

Bubble Sort Algorithm

- Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.
- **How Bubble Sort Works?**
- We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



- Bubble sort starts with very first two elements, comparing them to check which one is greater.

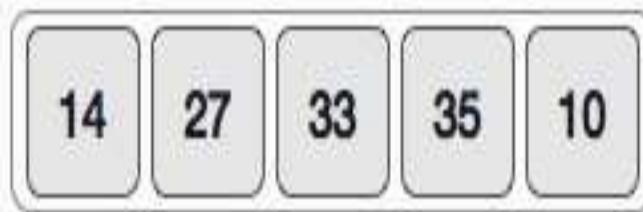


- In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

Bubble Sort Algorithm



We find that 27 is smaller than 33 and these two values must be swapped.



Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to the next two values, 35 and 10



We know then that 10 is smaller 35. Hence they are not sorted.



Bubble Sort Algorithm

- We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this



To be precise, we are now showing how an array should look like after each iteration.
After the second iteration, it should look like this



Notice that after each iteration, at least one value moves at the end



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Bubble Sort Algorithm

- **Algorithm**
 - We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.
 - ✓ begin BubbleSort(list)
 - ✓ for all elements of list
 - ✓ if list[i] > list[i+1]
 - ✓ swap(list[i], list[i+1])
 - ✓ end if
 - ✓ end for return list
 - ✓ end BubbleSortes
 - We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.
 - To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Insertion Sort Algorithm

6 5 3 1 8 7 2 4

Insertion Sort

➤ ALGORITHM

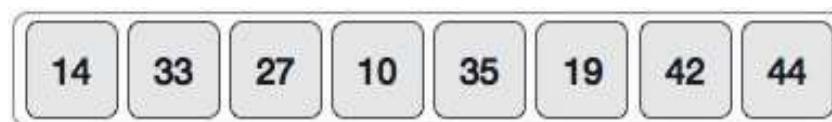
- ✓ **Step 1** – If it is the first element, it is already sorted.
return 1;
- ✓ **Step 2** – Pick next element
- ✓ **Step 3** – Compare with all elements in the sorted sub-list
- ✓ **Step 4** – Shift all the elements in the sorted sub-list that is greater than the value to be sorted
- ✓ **Step 5** – Insert the value
- ✓ **Step 6** – Repeat until list is sorted

Insertion Sort

- This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.
- The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

➤ How Insertion Sort Works?

- We take an unsorted array for our example.



- Insertion sort compares the first two elements



Insertion Sort

- It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



- Insertion sort moves ahead and compares 33 with 27.



- And finds that 33 is not in the correct position.



- It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



- By now we have t, it compares 33 with 10.



Insertion Sort

- These values are not in a sorted order.



- So we swap them.



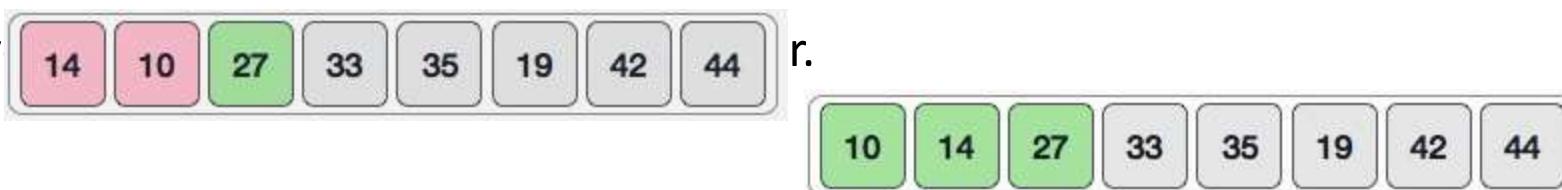
- However, swapping makes 27 and 10 unsorted.



- Hence, we swap again.



- Again we swap again.



SELECTION SORT ALGORITHM

8
5
2
6
9
3
1
4
0
7

Selection Sort

➤ Algorithm

✓ **Step 1** – Set MIN to location 0

✓ **Step 2** – Search the minimum element in the list

✓ **Step 3** – Swap with value at location MIN

✓ **Step 4** – Increment MIN to point to next element

✓ **Step 5** – Repeat until list is sorted

Selection Sort

- Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.
- This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

• How Selection Sort Works?

- Consider the following depicted array as an example.



- For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.



Selection Sort

- The same process is applied to the rest of the items in the array.
- Following is a pictorial depiction of the entire sorting process



MERGE SORT ALGORITHM

6 5 3 1 8 7 2 4

Merge Sort

➤ Algorithm

- Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted.
- Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

✓ **Step 1** – if it is only one element in the list it is already sorted, return.

✓ **Step 2** – divide the list recursively into two halves until it can no more be divided.

✓ **Step 3** – merge the smaller lists into new list in sorted order.

↳ Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being $O(n \log n)$, it is one of the most respected algorithms.

↳ Merge sort first divides the array into equal halves and then combines them in a sorted manner.

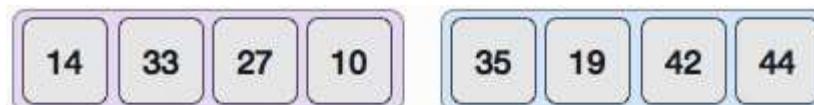
Merge Sort

➤ How Merge Sort Works?

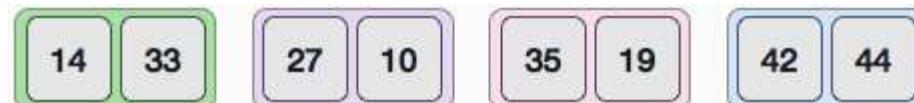
- To understand merge sort, we take an unsorted array as the following



We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

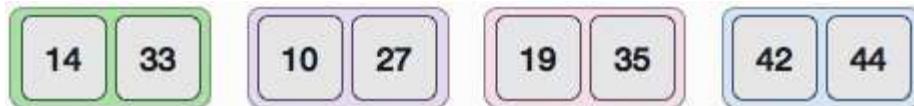


We further divide these arrays and we achieve atomic value which can no more be divided.



Merge Sort

- Now, we combine them in exactly the same manner as they were broken down. [Please note the color codes given to these lists.](#)
- We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



- In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this



SHELL SORT ALGORITHM

Shell Sort

- **Algorithm**

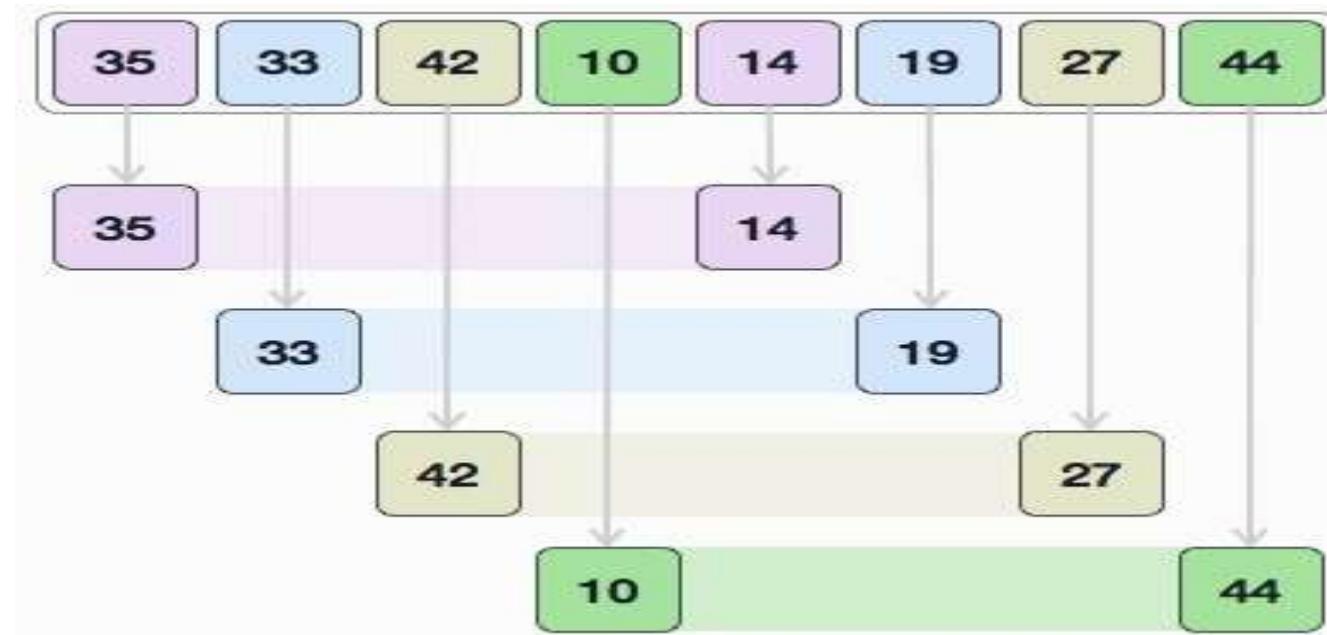
- ✓ **Step 1** – Initialize the value of h
- ✓ **Step 2** – Divide the list into smaller sub-list of equal interval h
- ✓ **Step 3** – Sort these sub-lists using **insertion sort**
- ✓ **Step 4** – Repeat until complete list is sorted
 - Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.
 - This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**. This interval is calculated based on Knuth's formula

➤ **Knuth's Formula**

- ✓ $h = h * 3 + 1$
- ✓ where – h is interval with initial value 1

Shell Sort

- This algorithm is quite efficient for medium-sized data sets as its average and worst case complexity are of $O(n)$, where n is the number of items.
- **How Shell Sort Works?**
- Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 14}

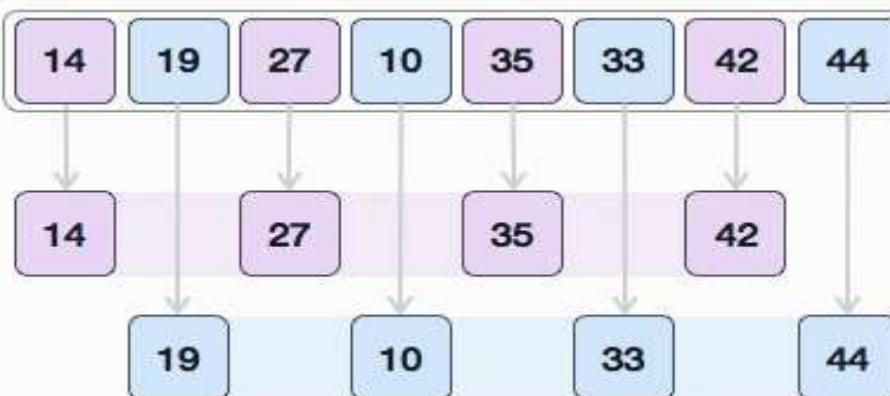


Shell Sort

- We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this



- Then, we take interval of 2 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this



Shell Sort

- Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.



We see that it required only four swaps to sort the rest of the array

QUICK SORT ALGORITHM

QUICK SORT

➤ Quick Sort Algorithm

It Is Sorted Quickly The Position Of The Element Is Replaced Quickly From Last Element

1. Start
2. Find The Position Of An Pivot Element
3. Let Us Consider The First Element As A Pivot Value.
4. Initialize I To Low & J To High Index's.
5. Repeat The Steps Until $I < j$
 - I. Keep On Increment The I Value While $A[i] \leq \text{pivot}$
 - II. Keep On Decrement The J Value While $A[j] > \text{pivot}$
 - III. If The Condition Is True [$I < j$] Swap $A[i], a[j]$
 - IV. If The Condition Is False [$I > j$] Swap $A[j], \text{Pivot}$
6. Now J Is The Exact Position Of The Pivot.
7. Stop

a[] =	0	1	2	3	4	5
	30	20	10	50	60	40

i

j

a[i]	<	a[j]
0	<	5

Find the I & J values

a[I] <= PIVOT &
i++

I value	30	\leq	30	20	\leq	30	10	\leq	30	50	\leq	30
	TRUE			TRUE			TRUE			FALSE		

i = 3

**a[j] > PIVOT &
J - -**

J value	40	>	30	60	>	30	50	>	30	10	>	30
	TRUE		TRUE		TRUE		TRUE		FALSE			

$$j = 2$$

- Now check the conditions
- if $i < j$ swap $a[i], a[j]$ - $3 < 2$ **False**
OR
- if $i > j$ swap $a[j], PIVOT$ - $3 > 2$ **True**

- Now swap the elements $a[j]$ & PIVOT
- After Swapping the elements are

10	20	30	50	60	40
----	----	----	----	----	----

- If the partition is having only 2 elements we are unable to sort the data

From The Second Part

50	60	40
i		j

	a[i]	<	a[j]	
I value	0	<	2	TRUE
	3	<	5	

a[I] <= PIVOT &

i++

50	60	40
----	----	----

i

j

i	=	1
---	---	---

50	<=	50	60	<=	50
TRUE			FALSE		

a[J] > PIVOT &

J --

J value	40	>	50
FALSE			

j	=	2
---	---	---

➤ Now check the conditions

➤ if I < J swap a [i], a[j]- 1 < 2 **True**

or

➤ if I > J swap a[j], PIVOT- 1 > 2 **False**

So swap a[I], a[J] then

50	40	60
----	----	----

NOTE:-

Now the pivot value is not changed so repeat the process until the pivot value swaps

HEAP SORT ALGORITHM

6 5 3 1 8 7 2 4

HEAP SORT

➤ ALGORITHM

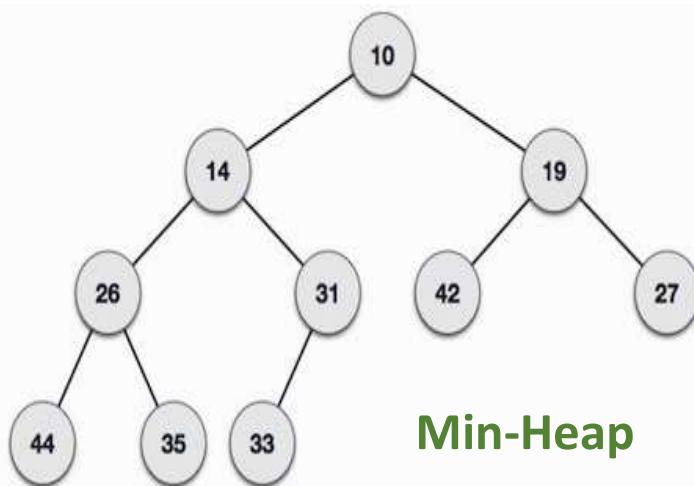
- ✓ **Step 1** – Create a new node at the end of heap.
- ✓ **Step 2** – Assign new value to the node.
- ✓ **Step 3** – Compare the value of this child node with its parent.
- ✓ **Step 4** – If value of parent is less than child, then swap them.
- ✓ **Step 5** – Repeat step 3 & 4 until Heap property holds.

➤ Max Heap Deletion Algorithm

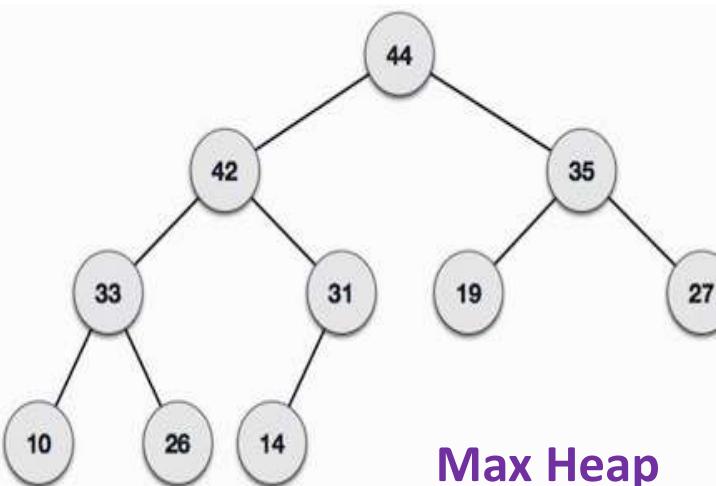
- ✓ **Step 1** – Remove root node.
- ✓ **Step 2** – Move the last element of last level to root.
- ✓ **Step 3** – Compare the value of this child node with its parent.
- ✓ **Step 4** – If value of parent is less than child, then swap them.
- ✓ **Step 5** – Repeat step 3 & 4 until Heap property holds.

HEAP SORT

- Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then
- As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types
- **For Input → 35 33 42 10 14 19 27 44 26 31**



Min-Heap



Max Heap

HEAP SORT

➤ Max Heap Construction Algorithm

- ✓ We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.
- ✓ We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.
- **Note** – In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.
- Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

HEAP SORT

➤ Max Heap Construction

Input 35 33 42 10 14 19 27 44 26 31

➤ Max Heap Deletion Algorithm

- Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

HEAP SORT

➤ Max Heap Deletion Algorithm

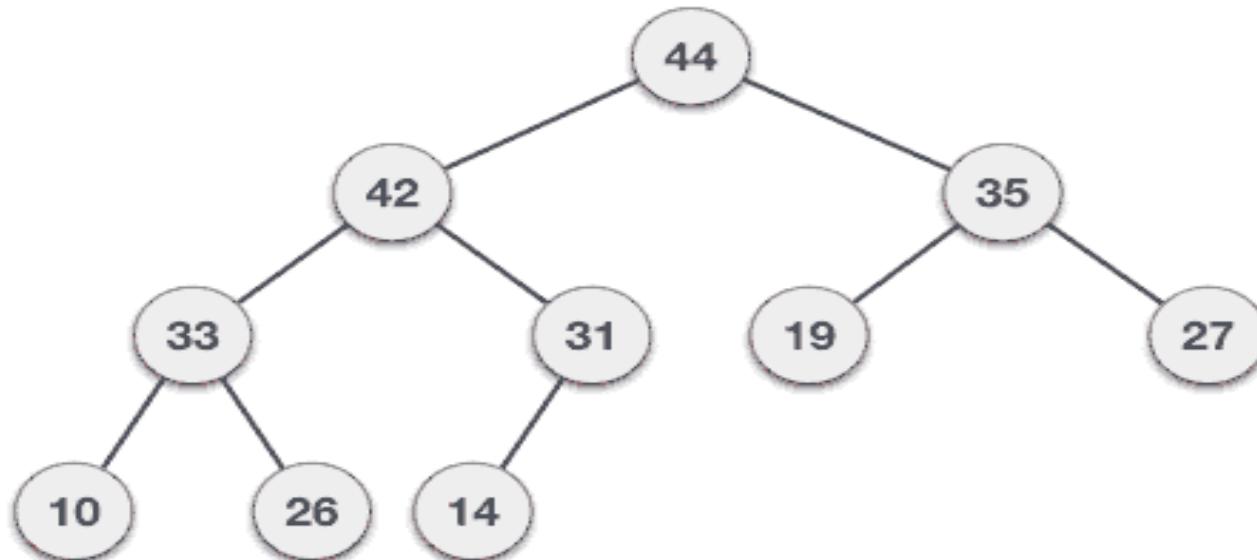
✓ **Step 1** – Remove root node.

✓ **Step 2** – Move the last element of last level to root.

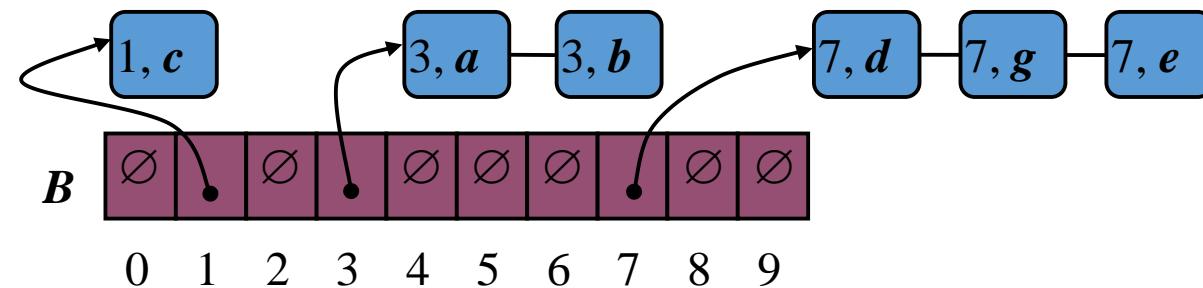
✓ **Step 3** – Compare the value of this child node with its parent.

✓ **Step 4** – If value of parent is less than child, then swap them.

✓ **Step 5** – Repeat step 3 & 4 until Heap property holds.



Radix/Bucket Sort



Radix Sort



- Radix sort is a linear sorting algorithm for integers that uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the radix is 26 (or 26 buckets) because there are 26 letters of the alphabet.
- Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begins with 'A', the second class contains names with 'B', so on and so forth.
- During the second pass, names are grouped according to the second letter. After the second pass, the names are sorted on the first two letters. This process is continued till nth pass, where n is the length of the names with maximum letters.
- After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains names beginning with 'A'. In the second pass collect the names from the second bucket, so on and so forth.

Radix Sort



- When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to most significant digit. When sorting numbers, we will have ten buckets, each for one digit (0, 1, 2..., 9) and the number of passes will depend on the length of the number having maximum digits.
- **Example:** Sort the numbers given below using radix sort.

345,654, 924, 123, 567, 472, 555, 808, 911

- In the first pass the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Radix Sort

- ✓ Step 1: Find the largest number in ARR as LARGE
- ✓ Step 2: [Initialize] SET NOP = Number of digits in LARGE
- ✓ Step 3: SET PASS = 0
- ✓ Step 4: Repeat Step 5 while PASS <= NOP-1
- ✓ Step 5: SET I = 0 AND Initialize buckets
- ✓ Step 6: Repeat Step 7 to Step 9 while I<N-1
- ✓ Step 7: SET DIGIT = digit at PASS th place in A[I]
- ✓ Step 8: Add A[I] to the bucket numbered DIGIT
- ✓ Step 9: INCREMENT bucket count for bucket numbered DIGIT
- ✓ Step 10: Collect the numbers in the bucket

Radix Sort



Radix Sort

Radix Sort

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567							567			
472					472					

After this pass, the numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as, **123, 345, 472, 555, 567, 654, 808, 911, 924**.

UNIT 5



Search



© Can Stock Photo - csp5895569



www.shutterstock.com - 90608047

Searching

➤ What is Search?

- Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Linear Search



Linear Search Algorithm (Sequential Search)

Searching

- ✓ Linear search algorithm finds given element in a list of elements with **O(n)** time complexity where **n** is total number of elements in the list.

- ✓ This search process starts comparing of search element with the first element in the list.

- ✓ If both are matching then results with element found otherwise search element is compared with next element in the list.

- ✓ If both are matched, then the result is "element found".

- ✓ Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list".

- ✓ That means, the search element is compared with element by element in the list.

Searching

➤ Algorithm

- ✓ **Step 1:** Read the search element from the user
- ✓ **Step 2:** Compare, the search element with the first element in the list.
- ✓ **Step 3:** If both are matching, then display "Given element found!!!" and terminate the function
- ✓ **Step 4:** If both are not matching, then compare search element with the next element in the list.
- ✓ **Step 5:** Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- ✓ **Step 6:** If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Searching

➤ Example

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
search element	12							

Step 1:

search element (12) is compared with first element (65)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
	12							

Both are not matching. So move to next element

Searching

Step 2:

search element (12) is compared with next element (20)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
	12							

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
	12							

Both are not matching. So move to next element

Searching

Step 4:

search element (12) is compared with next element (55)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
	12							

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

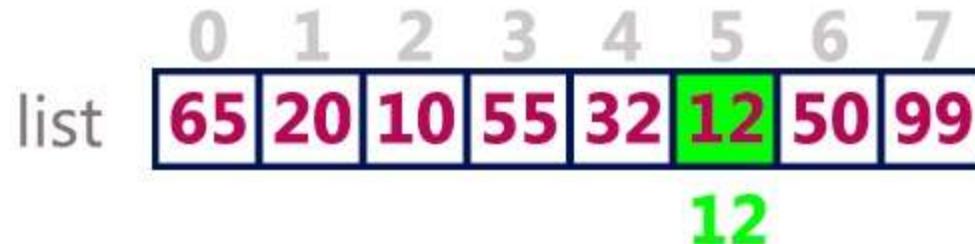
	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99
	12							

Both are not matching. So move to next element

Searching

Step 6:

search element (12) is compared with next element (12)



Both are matching. So we stop comparing and display element found at index 5.

Binary Search Algorithm

Binary Search

- ✓ Binary search algorithm finds given element in a list of elements with **O(log n)** time complexity where **n** is total number of elements in the list.
- ✓ The binary search algorithm can be used with only sorted list of element.
- ✓ That means, binary search can be used only with list of element which are already arranged in a order.
- ✓ The binary search can not be used for list of element which are in random order. This search process starts comparing of the search element with the middle element in the list.
- ✓ If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list.
- ✓ If the search element is smaller, then we repeat the same process for left sublist of the middle element.
- ✓ If the search element is larger, then we repeat the same process for right sublist of the middle element.
- ✓ We repeat this process until we find the search element in the list or until we left with a sublist of only one element.
- ✓ And if that element also doesn't match with the search element, then the result is "Element not found in the list".

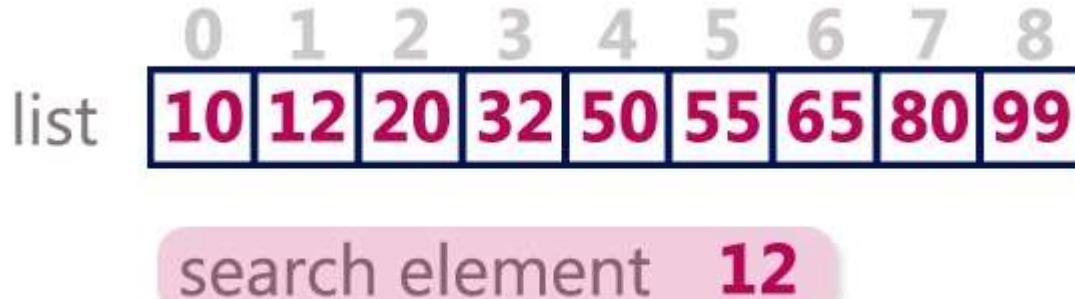
Binary Search

➤ Algorithm

- ✓ **Step 1:** Read the search element from the user
- ✓ **Step 2:** Find the middle element in the sorted list
- ✓ **Step 3:** Compare, the search element with the middle element in the sorted list.
- ✓ **Step 4:** If both are matching, then display "Given element found!!!" and terminate the function
- ✓ **Step 5:** If both are not matching, then check whether the search element is smaller or larger than middle element.
- ✓ **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- ✓ **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- ✓ **Step 8:** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- ✓ **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

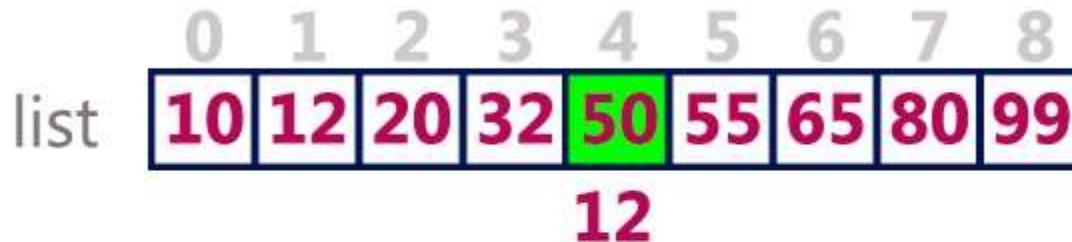
Binary Search

➤ Example



Step 1:

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).



Binary Search

Step 2:

search element (12) is compared with middle element (12)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Both are matching. So the result is “Element found at index 1”

search element 80

Step 1:

search element (80) is compared with middle element (50)

list [10, 12, 20, 32, 50, 55, 65, 80, 99]
80

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

list	0	1	2	3	4	5	6	7	8
	10	12	20	32	50	55	65	80	99

Binary Search

Step 2:

search element (80) is compared with middle element (65)

list	0	1	2	3	4	5	6	7	8
	10	12	20	32	50	55	65	80	99
							80		

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

list	0	1	2	3	4	5	6	7	8
	10	12	20	32	50	55	65	80	99
							80	99	

Step 3:

search element (80) is compared with middle element (80)

list	0	1	2	3	4	5	6	7	8
	10	12	20	32	50	55	65	80	99
							80	99	

Both are not matching. So the result is "Element found at index 7"

HASH TABLES

In Data Structures,

- There are several searching techniques like linear search, binary search, search trees etc.
- In these techniques, time taken to search any particular element depends on the total number of elements.

Example-

- Linear Search takes $O(n)$ time to perform the search in unsorted arrays consisting of n elements.
- Binary Search takes $O(\log n)$ time to perform the search in sorted arrays consisting of n elements.
- It takes $O(\log n)$ time to perform the search in Binary Search Tree consisting of n elements.

Drawback

The main drawback of these techniques is

- As the number of elements increases, time taken to perform the search also increases.
- This becomes problematic when total number of elements become too large.

HASHING

- Hashing is a well-known technique to search any particular element among several elements.
- It minimizes the number of comparisons while performing the search.

ADVANTAGE

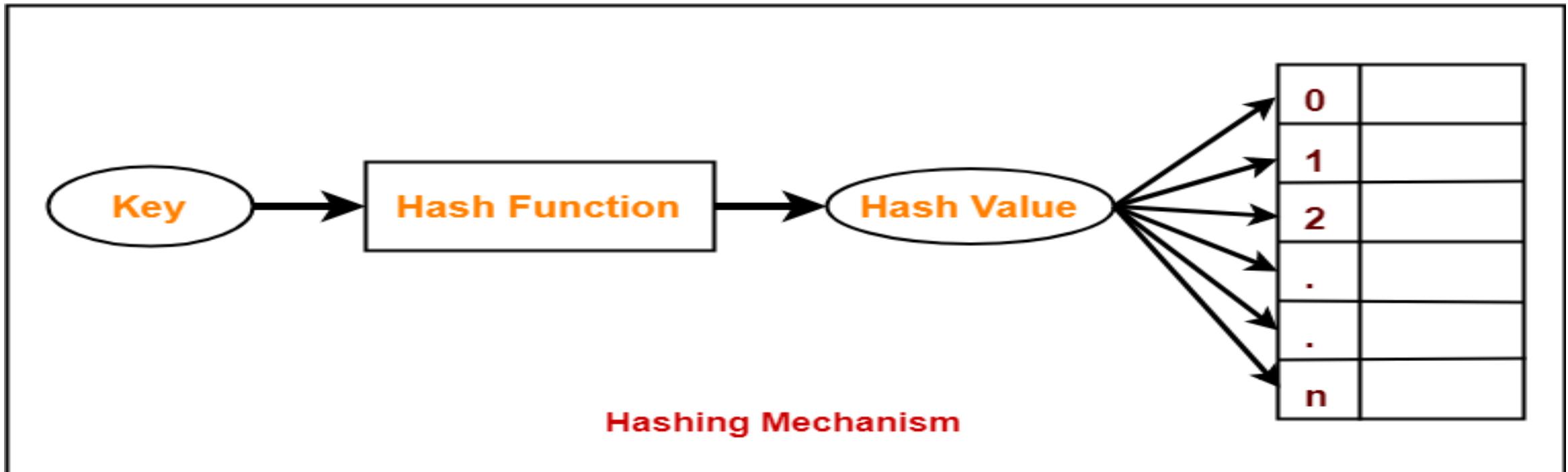
Unlike other searching techniques,

- Hashing is extremely efficient.
- The time taken by it to perform the search does not depend upon the total number of elements.
- It completes the search with constant time complexity $O(1)$.

HASHING MECHANISM

In hashing,

- An array data structure called as Hash table is used to store the data items.
- Based on the hash key value, data items are inserted into the hash table.



TYPES OF HASHING

There are two types of hashing.

1. Static hashing
2. Dynamic hashing

STATIC HASHING

Static hashing is a hashing technique in which keys are stored in hash table with fixed size.

DYNAMIC HASHING

In this hashing table, the hash function is modified dynamically as number of records grow.

HASH KEY VALUE

- Hash key value is a special value that serves as an index for a data item.
- It indicates where the data item should be stored in the hash table.
- Hash key value is generated using a hash function.

HASH FUNCTION

Hash function is a function that maps any big number or string to a small integer value.

- Hash function takes the data item as an input and returns a small integer value as an output.
- The small integer value is called as a hash value.
- Hash value of the data item is then used as an index for storing it into the hash table.

TYPES OF HASH FUNCTIONS

- Mid Square Hash Function
- Division Hash Function
- Folding Hash Function etc

“It depends on the user which hash function he wants to use.”

PROPERTIES OF HASH FUNCTION

The properties of a good hash function are

- It is efficiently computable.
- It minimizes the number of collisions.
- It distributes the keys uniformly over the table.

Buck and Home bucket

The hash function $H(key)$ is used to map several dictionary entries in the hash table. Each function of hash table is called bucket. The function $H(k)$ is home bucket for the dictionary with pair whose value is key.

KEY VALUE	INDEX
	5
15	4
	3
10	2
	1
	0

HASH TABLE

In the above diagram or hash table location 2 or 4 is called as home bucket and location 0,1,3,5 are called as bucket.

Division hash function method

The hash function depends upon the remainder of the division. Typically the division is the table length.

SYNTAX OR FORMULA

$$H(\text{key}) = K \% \text{table size}$$

EXAMPLE:-

Insert following values or records

54, 72, 89, 37 into hash table. The hash table size is 10.

The record **54** is inserted into above hash table by using division hash function.

$$H(\text{key}) = k \% \text{table size}$$

$$H(\text{Key}) = 54 \% 10 = 4$$

	9
	8
	7
	6
	5
	4
	3
	2
	1
	0

The record **54** is inserted at **4th** location.

The record 72 is inserted into above hash table by using division hash function.

$$\begin{aligned}H(\text{key}) &= \text{k \% table size} \\H(\text{Key}) &= 72 \% 10 = 2\end{aligned}$$

The record **72** is inserted at **2nd** location.

The record 89 is inserted into above hash table by using division hash function.

$$\begin{aligned}H(\text{key}) &= \text{k \% table size} \\H(\text{Key}) &= 89 \% 10 = 9\end{aligned}$$

The record **89** is inserted at **9th** position or location.

The record 37 is inserted into above hash table by using division hash function.

$$\begin{aligned}H(\text{key}) &= \text{k \% table size} \\H(\text{Key}) &= 37 \% 10 = 7\end{aligned}$$

The record **37** is inserted at **7th** position.

The following hash table determines the inserting records 54, 72, 89, 37 into hash table.

89	9
	8
37	7
	6
	5
54	4
	3
72	2
	1
	0

MID SQUARE HASH FUNCTION

In the mid square method, the key is squared and the middle or mid part of the result is used as index or position or location.

Example the records 311, 3112, 3114 are inserted to hash table. Assume that hash table size is 1000.

SYNTAX OR FORMULA

$$H(\text{Key}) = K^2$$

EXAMPLE:-

The record 3111 by using mid square.

$$H(\text{key}) = K^2$$

$$= (3111)^2$$

$$= 96\textcolor{red}{783}21$$

783 is the middle part of **9678321**. So, 783 is the index of 3111.

The record 3112 by using mid square

$$H(\text{Key}) = (3112)^2$$

$$= 96\mathbf{845}44$$

845 is the middle part of **9684544**. So, 845 is the index of 3112.

The record 3113 by using mid square

$$H(\text{Key}) = (3113)^2$$

$$= 96\mathbf{907}69$$

907 is the middle part of **9690769**. So, 907 is the index of 3113.

3111	783
3112	845
3113	907
	999

MULTIPLICATIVE HASH FUNCTION

The given record is multiplied by some constant value. The formula computing hash key is

$$H(\text{Key}) = \text{floor}(P * (\text{fractional part of key} * A))$$

Where ‘P’ is an integer constant and ‘A’ is real constant.

Donald Knuth suggested to use constant $A = 0.61803398987$.

EXAMPLE:

Insert the following records **107, 108, 109, 110** into hash table . Here $P = 50$.

107 inserted into hash table by using multiplicative hash function.

$$H(\text{Key}) = \text{floor}(P * (\text{fractional part of key} * A))$$

$$= \text{floor}(50 * (107 * 0.61803398987))$$

$$= \text{floor}(3306.4818)$$

$$= 3306$$

108 inserted into hash table by using multiplicative hash function.

$$\begin{aligned} H(\text{Key}) &= \text{floor}(50 * (108 * 0.61803398987)) \\ &= \text{floor}(3337.3835) \\ &= 3337 \end{aligned}$$

109 inserted into hash table by using multiplicative hash function.

$$\begin{aligned} H(\text{Key}) &= \text{floor}(50 * (109 * 0.61803398987)) \\ &= \text{floor}(3368.2852) \\ &= 3368 \end{aligned}$$

110 inserted into hash table by using multiplicative hash function.

$$\begin{aligned} H(\text{Key}) &= \text{floor}(50 * (110 * 0.61803398987)) \\ &= \text{floor}(3399.1869) \\ &= 3399 \end{aligned}$$

	0
107	3306
108	3337
109	3368
110	3399
	3999

DIGIT FOLDING OR FOLDING HASH FUNCTION

The key value is divided into separate parts and using some simple operation this parts are combined to produce hash key.

EXAMPLE:

Consider the record 1, 2, 3, 6, 5, 4, 1, 2 then it is divided into separate parts 123, 654, 12 and this all are added together.

$$H(\text{Key}) = 123 + 654 + 12 + 789$$

The record 123, 654, 12 will be placed at a location 789 in the hash table.

COLLISION RESOLUTION TECHNIQUE

- Hashing is a well-known searching technique.
- It minimizes the number of comparisons while performing the search.
- It completes the search with constant time complexity $O(1)$.

COLLISION

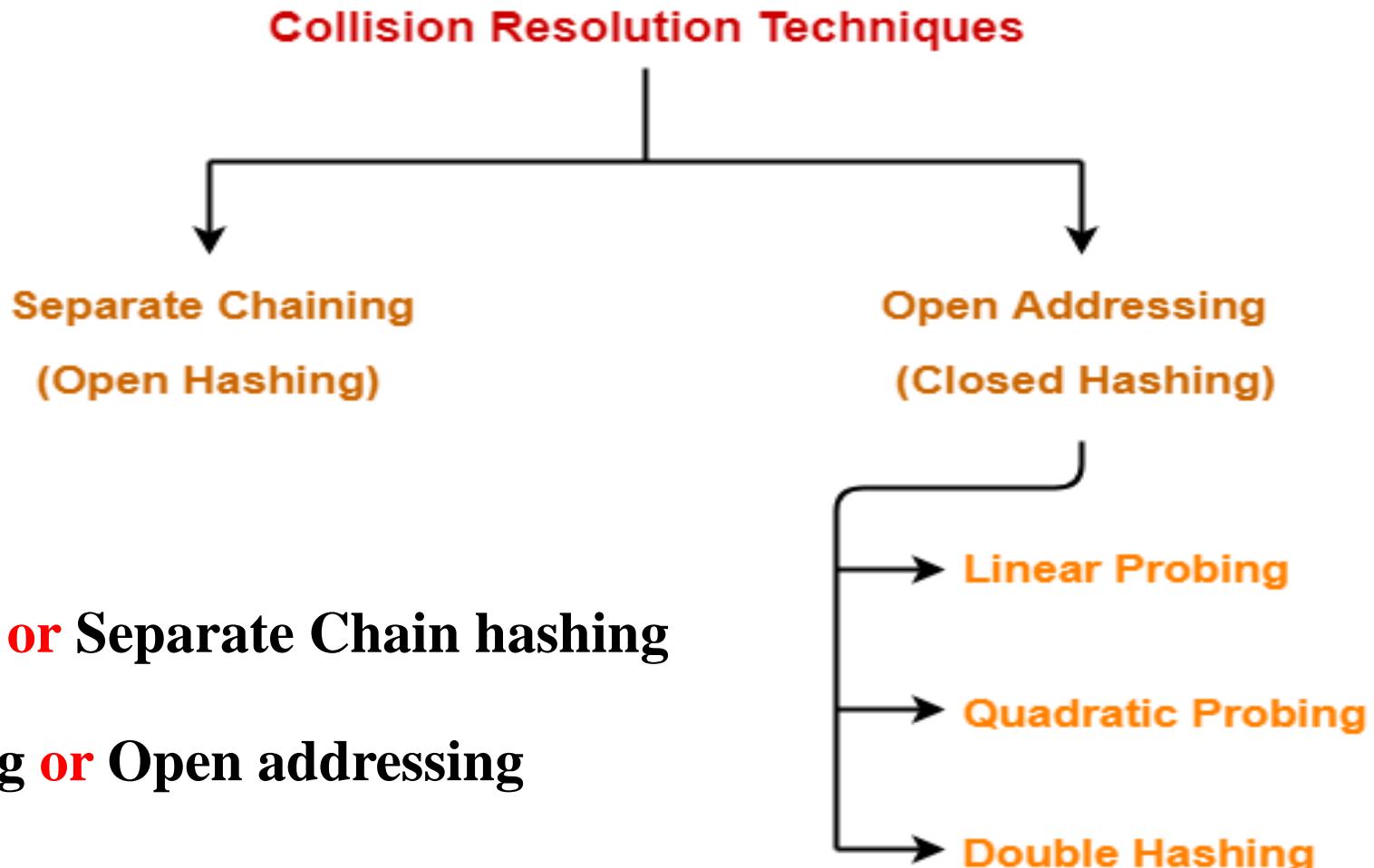
When the hash value of a key maps to an already occupied bucket of the hash table, it is called as a Collision.

If collision occurs then it should be handled by applying some techniques. Such techniques are called collision resolution technique.

The goal of collision resolution techniques is to minimize collisions. There are two methods of handling collisions.

COLLISION RESOLUTION TECHNIQUES

Collision Resolution Techniques are the techniques used for resolving or handling the collision



1. Open hashing **or** Separate Chain hashing
2. Closed hashing **or** Open addressing

SEPARATE CHAINING

OR

OPEN HASHING

SEPARATE CHAINING OR OPEN HASHING

To handle the collision,

- This technique creates a linked list to the slot for which collision occurs.
- The new key is then inserted in the linked list.
- These linked lists to the slots appear like chains.
- That is why, this technique is called as separate chaining.

Time Complexity For Searching

- In worst case, all the keys might map to the same bucket of the hash table.
- In such a case, all the keys will be present in a single linked list.
- Sequential search will have to be performed on the linked list to perform the search.
- So, time taken for searching in worst case is $O(n)$.

For Deletion

- In worst case, the key might have to be searched first and then deleted.
- In worst case, time taken for searching is $O(n)$.
- So, time taken for deletion in worst case is $O(n)$.

Load Factor (α)

Load factor (α) is defined as

If Load factor (α) = constant, then time complexity of Insert, Search, Delete = $\Theta(1)$

$$\text{Load Factor } (\alpha) = \frac{\text{Number of elements present in the hash table}}{\text{Total size of the hash table}}$$

PRACTICE PROBLEM BASED ON SEPARATE CHAINING

Using the hash function ‘key mod 7’, insert the following sequence of keys in the hash table.

50, 700, 76, 85, 92, 73 and 101

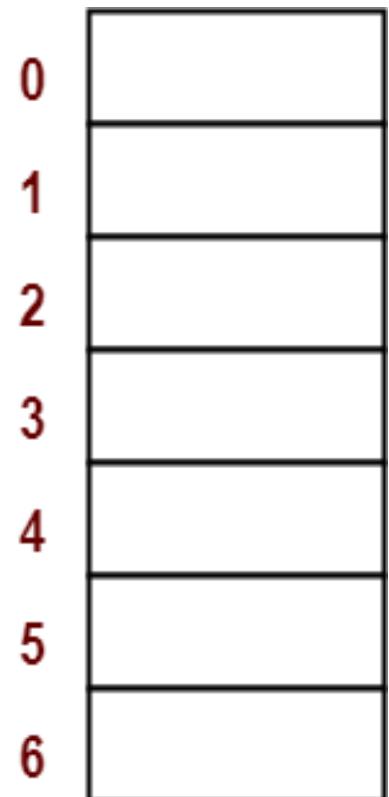
Use separate chaining technique for collision resolution.

SOLUTION

The given sequence of keys will be inserted in the hash table as-

STEP-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as.



STEP-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = $50 \bmod 7 = 1$.
- So, key 50 will be inserted in bucket-1 of the hash table as.

0	
1	50
2	
3	
4	
5	
6	

STEP-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = $700 \bmod 7 = 0$.
- So, key 700 will be inserted in bucket-0 of the hash table as.

0	700
1	50
2	
3	
4	
5	
6	

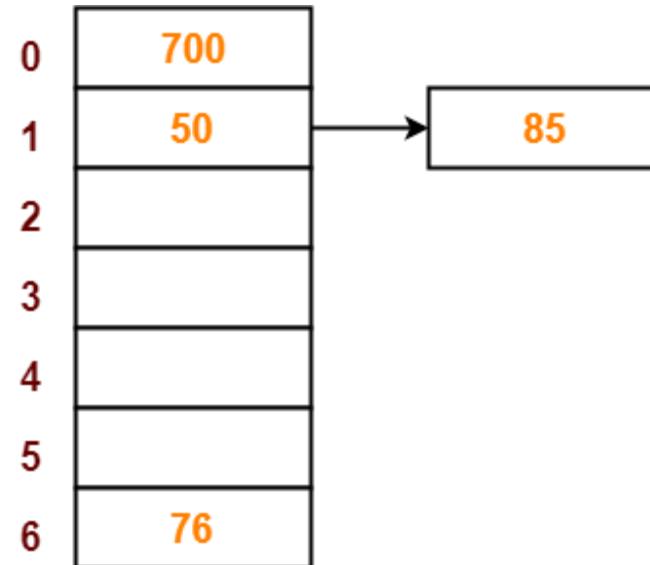
STEP-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = $76 \bmod 7 = 6$.
- So, key 76 will be inserted in bucket-6 of the hash table as.

0	700
1	50
2	
3	
4	
5	
6	76

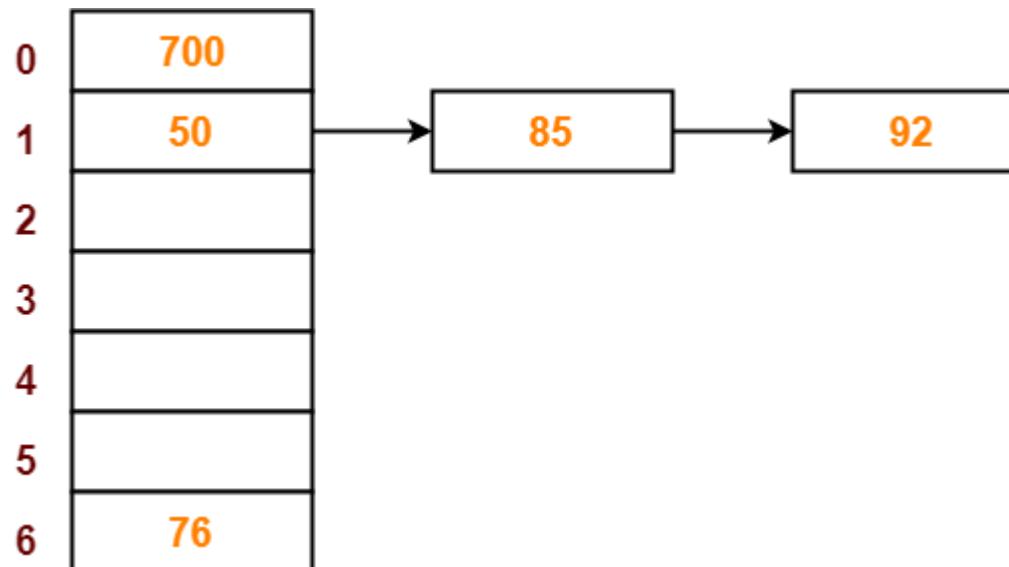
STEP-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = $85 \bmod 7 = 1$.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 85 will be inserted in bucket-1 of the hash table as.



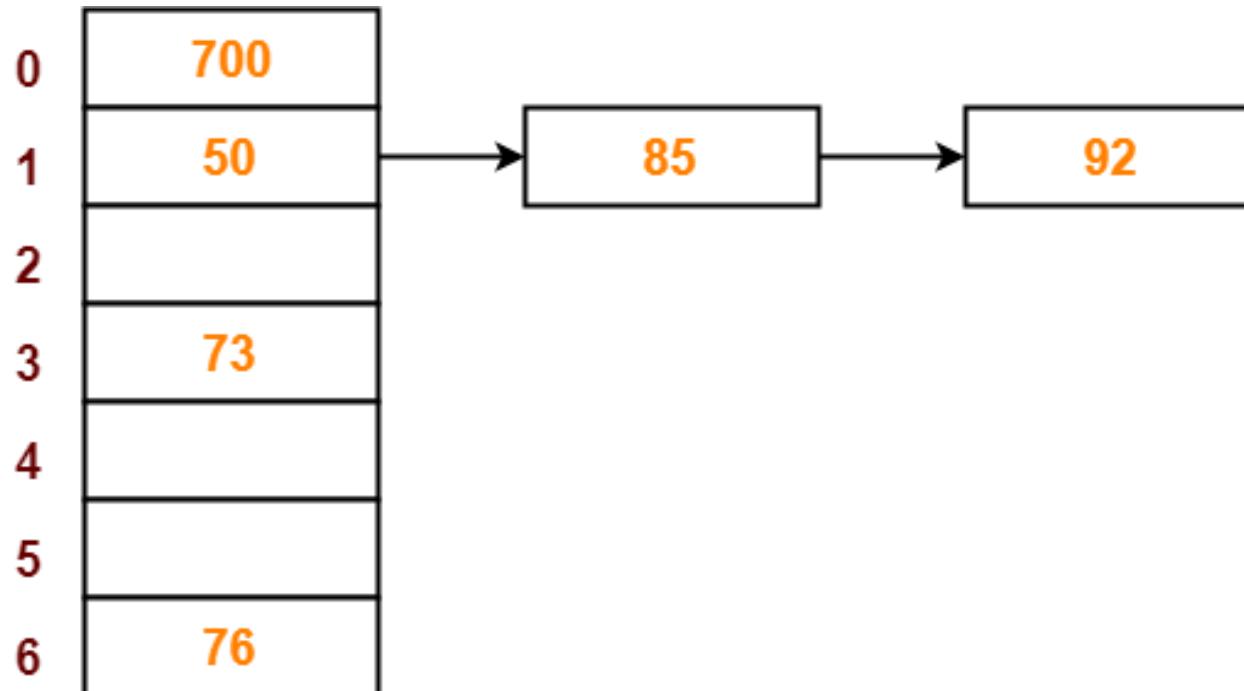
STEP-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = $92 \bmod 7 = 1$.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 92 will be inserted in bucket-1 of the hash table as.



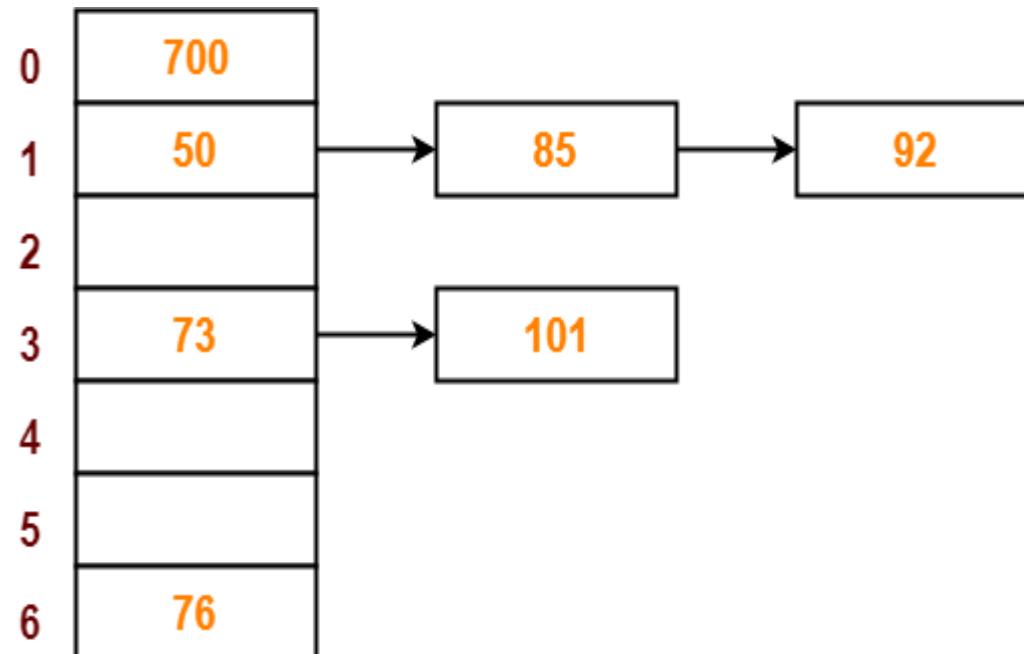
STEP-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = $73 \bmod 7 = 3$.
- So, key 73 will be inserted in bucket-3 of the hash table as.



STEP-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = $101 \bmod 7 = 3$.
- Since bucket-3 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-3.
- So, key 101 will be inserted in bucket-3 of the hash table as.



**OPEN ADDRESSING
OR
CLOSE HASHING**

IN OPEN ADDRESSING,

- Unlike separate chaining, all the keys are stored inside the hash table.
- No key is stored outside the hash table

TECHNIQUES USED FOR OPEN ADDRESSING:

- Linear Probing
- Quadratic Probing
- Double Hashing

OPERATIONS IN OPEN ADDRESSING

- Insert Operation
- Search Operation
- Delete Operation

INSERT OPERATION

- Hash function is used to compute the hash value for a key to be inserted.
- Hash value is then used as an index to store the key in the hash table.

IN CASE OF COLLISION,

- Probing is performed until an empty bucket is found.
- Once an empty bucket is found, the key is inserted.
- Probing is performed in accordance with the technique used for open addressing.

SEARCH OPERATION

To search any particular key,

- Its hash value is obtained using the hash function used.
- Using the hash value, that bucket of the hash table is checked.
- If the required key is found, the key is searched.
- Otherwise, the subsequent buckets are checked until the required key or an empty bucket is found.
- The empty bucket indicates that the key is not present in the hash table.

DELETE OPERATION

- The key is first searched and then deleted.
- After deleting the key, that particular bucket is marked as “deleted”.

OPEN ADDRESSING TECHNIQUES

1. Linear Probing

- When collision occurs, we linearly probe for the next bucket.
- We keep probing until an empty bucket is found.

ADVANTAGE

- It is easy to compute.

DISADVANTAGE

- The main problem with linear probing is clustering.
- Many consecutive elements form groups.
- Then, it takes time to search an element or to find an empty bucket.

TIME COMPLEXITY

Worst time to search an element in linear probing is $O(n)$ (table size).

This is because

- Even if there is only one element present and all other elements are deleted.
- Then, “deleted” markers present in the hash table makes search the entire table.

EXAMPLE:

Consider that following keys are to be inserted in the hash table 131, 4, 8, 7, 21, 5, 31, 61, 9, 29. The hash table size is 10.

Initially we will put the following keys in the hash table 131, 4, 8, 7.

We will use division hash function. That means that keys are placed using formula.

$$H(\text{Key}) = \text{key \% table size}$$

For instance the element 131 can be placed at $H(\text{Key}) = 131 \% 10 = 1$.

Index 1 will be the home bucket for 131. Continuing in the fashion we will place 4,8,7.

0	Null
1	131
2	Null
3	Null
4	4
5	Null
6	Null
7	7
8	8
9	Null

Now the next to be inserted is 21. According to hash function

$$H(\text{Key}) = 21 \% 10 = 1.$$

- But the index 1 location already occupied with 131 i.e., collision occurs. To resolve this collision **we will linearly move down from 1 to empty location is found.**
- Therefore 21 will be placed at index 2.
- If the next element is 5 then we get home bucket for 5 as index 5 this bucket is empty so, we will put the element 5 at index 5.

0	Null
1	131
2	21
3	Null
4	4
5	5
6	Null
7	7
8	8
9	Null

After placing record keys **31**, **61** the hash table will be

0	Null
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	Null

The next record key that comes is **9**. According to decision as function it demands for the home bucket **9**. Hence we will place **9** at index **9**.

0	Null
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	9

0	29
1	131
2	21
3	31
4	4
5	5
6	61
7	7
8	8
9	9

- Now the next final record key is **29** and it hashes a key **9**. But home bucket **9** is already occupied. And there is no next empty bucket as the table size is limited to index **9**. The overflow occurs to handle it we move back to bucket **0** and as the location over there is empty **29** will be placed at **0th** index.

QUADRATIC PROBING

Quadratic probing operates by taking original hash value and adding successive values of quadratic polynomial to the starting value.

This method uses following formula.

$$H(Key) = (H(Key) + i^2) \% m$$

Where 'm' can be table size or any prime number.

EXAMPLE: - If we have insert following elements in the hash table with table **size 10**.

37, 19, 55, 22, 17, 49, 87.

Initially we will put following keys into hash table.

37, 19, 55, 22

0	
1	
2	22
3	
4	
5	55
6	
7	37
8	
9	

Now, if you want to place 17 a collision will be occurs 17. $17 \% 10 = 7$, but bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

$$H(\text{Key}) = (H(\text{Key}) + i^2) \% m$$

Consider I = 0

$$H(\text{key}) = (17 + 0^2) \% 10 = 17 \% 10 = 7.$$

Then i=1

$$H(\text{Key}) = (17 + 1^2) \% 10 = 18 \% 10 = 8.$$

The bucket 8 is empty. Hence we will place the element of the index 8.

0	49
1	
2	22
3	
4	
5	55
6	
7	37
8	17
9	19

Now if you want to place 49 a collision will be occur $49 \% 10 = 9$ and bucket 9 as already occupied with 19. Hence we will applying quadratic probing to insert this record in the hash table.

$$H_i(\text{Key}) = (H(\text{Key}) + i^2) \% m$$

$$I = 0 \quad = (49 + 0) \% 10 = 49 \% 10 = 9$$

➤ The bucket 0 is empty.

$$I = 1 \quad = (49 + 1^2) \% 10 = 50 \% 10 = 0$$

➤ Hence the value 49 is inserted at a 0th position.

Now to place 87 we will use quadratic probing.

$$H(Key) = (87 + 0^2) \% 10 = 87 \% 10 = 7$$

$$H(Key) = (87 + 1^2) \% 10 = 88 \% 10 = 8$$

$$H(Key) = (87 + 2^2) \% 10 = 91 \% 10 = 1$$

0	49
1	87
2	22
3	
4	
5	55
6	
7	37
8	17
9	19

DOUBLE PROBING

OR

DOUBLE HASHING

Double hashing is a technique in which a second hash function is applied to key when a collision occur by applying the second has function we will get number of positions from the point of Collision inserted.

By using following formulas we can find out the double hashing.

$$H_1(\text{key}) = k \% \text{ table size}$$

$$H_2(\text{key}) = M - (K \% M)$$

$$37 \% 10 = 7$$

$$90 \% 10 = 0$$

$$45 \% 10 = 5$$

$$22 \% 10 = 2$$

Where M is prime number smaller than the size of the table.

Example: consider the following elements to be placed in the Hash table of size 10.

37, 90, 45, 22, 17, 49, 55.

Inside Initially the elements using the formula for $H_1(\text{key})$. Insert 37, 90, 45, 22.

0	90
1	
2	22
3	
4	
5	45
6	
7	37
8	
9	

NOW IF 17 IS TO BE INSERTED THEN

$$H_1(17) = 17 \% 10 = 7$$

Here collision will be occur because 7th position already occupied with element 37 or record 37.
So we can apply second hash function to key.

$$H_2(\text{key}) = M - (\text{K} \% M)$$

Here M is prime number smaller than the size of the table.

Let us prime number is M = 7

$$H_2(17) = 7 - (17 \% 7)$$

$$= 7 - 3 = 4 \text{ (4 jumps from the collision index)}$$

17 will be placed at index 1.

0	90
1	17
2	22
3	
4	
5	45
6	
7	37
8	
9	

Now to insert number 49 at location 9th position that is $49 \% 10 = 9$.

0	90
1	17
2	22
3	
4	
5	45
6	
7	37
8	
9	49

Now to insert number 55.

$H_1(55) = 55 \% 10 = 5$ that is collision will be occur. Because the location 5 already occupied with 45. So, we can apply second hash function.

$$H_2(55) = 7 - (55 \% 7) = 7 - 6 = 1 \text{ (1 jumps from the collision index)}$$

That means we have to take one jump from index 5 to place 55.

0	90
1	17
2	22
3	
4	
5	45
6	55
7	37
8	
9	49

SEPARATE CHAINING	OPEN ADDRESSING
➤ Keys are stored inside the hash table as well as outside the hash table.	➤ All the keys are stored only inside the hash table. ➤ No key is present outside the hash table.
➤ The number of keys to be stored in the hash table can even exceed the size of the hash table.	➤ The number of keys to be stored in the hash table can never exceed the size of the hash table.
➤ Deletion is easier.	➤ Deletion is difficult.
➤ Extra space is required for the pointers to store the keys outside the hash table.	➤ No extra space is required.
➤ Cache performance is poor. ➤ This is because of linked lists which store the keys outside the hash table.	➤ Cache performance is better. ➤ This is because here no linked lists are used.
➤ Some buckets of the hash table are never used which leads to wastage of space.	➤ Buckets may be used even if no key maps to those particular buckets

Rehashing

Rehashing is a technique in which table is resized that is the size of table is double by creating a new table. It is preferable if the total size of new table is a prime number. There are situations in which rehashing is required.

- i) When the table size is completely full.
- ii) With Quadratic probing when the table is filled half.
- iii) When insertion fails due to overflow.

In such situations, we have to transfer entries from old table to new table.

THANK YOU